Universitetet i Oslo
Institutt for informatikk

Objektorientert Dokumentdesign i XML — Obdok

Hans Ole Gjerdrum

Hovedfagsoppgave

28. april 2003

Forord

Hovedfagsoppgaven som nå ligger for din hånd, er skrevet som en del av undertegnedes *Candidates Scientiarum*-grad innen studieretningen Informasjonsdesign (ID), Institutt for informatikk, Det matematisk-naturvitenskapelige fakultet ved Universitetet i Oslo

For at denne oppgaven har kunne latt seg realisere er jeg dypt takknemlig til en rekke personer. Først og fremst vil rette en stor takk til min aktede veileder Dino Karabeg for hans bidrag. Du har på en fantastisk måte motivert meg både gjennom en utfordrende oppgaveutforming og gjennom kyndig veiledning. At jeg aldri har fått den store "knekken" eller vært nede i de dype bølgedaler skal du ha mye av æren for. Din veiledning har dessuten ført frem mot en særs interessant og spennende prosess med dette skriv og tilhørende kode som resultat. Dette har også tilført meg mye nyttig kunnskap innen informasjonsdesign, hvilke utfordringer dette fagfeltet møter, og selvsagt innenfor arbeidet med **XML**-relaterte verktøy.

Videre må jeg takke min familie som alltid har stått meg bi og støttet meg gjennom hovedfag og studier. Mors kjøttkaker har reddet flere hovedfag enn hun aner, og det er utrolig godt å ha en far det er mulig å diskutere fag med.

Anna og Gunnhild: dere begge står meg nær og jeg er ytterst takknemlig for våre utenomfaglige diskusjoner, middager og “ekskursjoner”. Takk også til *Parken forever* for et utrolig godt miljø, og til Audun for gjennomlesning og kritikk. Hadde jeg ikke hatt Ingrid til å hjelpe meg med språk, retting og motivasjon, ville det hele føltes mye tyngre.

Til slutt vil jeg rette en stor takk til Håvard. Uten deg ville intet vært mulig.

Hans Ole Gjerdrum
28. april 2003

Innhold

Definisjoner

Kapittel 1

Introduksjon

«I begynnelsen var Ordet». Slik begynner evangelisten Johannes sin skapelsesberetning. *'Ordet'* stammer fra det greske 'Logos', som betyr ord, tanke eller fornuft (Lundeby 1984). Vår verden er altså tuftet på forstand, orden, kunnskap og fornuft. Da Sokrates uttalte at *«Jeg vet at jeg intet vet»* ble han utpekt av Orakelet i Delphi som den viseste av de vise. Han skjønte han befant seg i en verden full av kunnskap — både menneskelige ervervet og potensiell kunnskap — og at av dette var der bare brøkdeler Sokrates selv hadde tilegnet seg. Siden den tid, i vitenskapens barndom, har kunnskapens tre bare vokst og vokst. Dets grener har bredd seg utover et ubegripelig stort landskap og dets krone har fått en så tett, kompleks og ugjennomtrengelig form at menneske ikke lenger klarer nå dets frukter. Kanskje er det derfor heller så at denne historie burde starte som den greske mytologiens utgangspunkt: *«Først var der bare kaos»*? At vi befinner oss i et kaotisk univers av informasjon. En slik tilnærming vil også gjøre målsetningen for denne oppgaven klarere; nemlig hvordan forbedre kvaliteten og tilgjengeligheten til informasjon.

1.1 In Medias Res

Arbeidet med informasjonsforedling står i dag på et tilsvarende sted som utviklingen av programmeringsspråkene gjorde på 50- og 60-tallet. Da fantes kun simple imperative programmeringsspråk bestående av frittstående *primitive datatyper*, som *integer*, *float/real* eller *character*, eller enkle *komplekse datatyper* i form av *array*-konstruksjoner. I tidlige tider hadde språkene heller ingen midler for strukturering av data og hendelsesflyt, i metoder eller prosedyrer. Utsagnene ble således eksekvert direkte, sekvensielt og inkrementelt, eller ved testing og GoTo'er. Mellom dette og tradisjonell fortellerteknikk, både muntlig og skriftlig,

finner man en slående likhet. Der finnes de samme typer enkle “datatypene” i ord og setninger (arrayer av ord) og den samme strenge sekvensielle og inkrementelle — "perm til perm" — gjennomløping av fortellingene.

Til tross for at den moderne tid har brakt frem nyvinninger for mer effektiv og innovativ fremlegging av informasjon, blir ikke alltid disse utnyttet til fulle. Snarere benyttes fremdeles gamle og tradisjonelle utviklingsmønstre under slike prosesser — mønstre som stammer tilbake i fra oral fortellerteknikk og Gutenbergs boktrykkermaskin. Den lineære og flate orale fremstillingsmåten preger fremdeles informasjonen, selv om moderne teknologier som elektroniske bøker, Internett og *hypermedia*, blir benyttet til spredning og søk etter denne. Resultatet av dette fortoner seg ofte som rotete og ustrukturerte for både leser så vel som forfatter. I hypermediadokumenter ligner linker på mange måter GoTo’ene som ble brukt i programmeringen i forgangene dager.

Med målsetting om å kunne øke lesbarheten, samt å organisere og strukturere programmer på en mer effektiv måte, ble konseptene bak objektorientert programmering utviklet på 1960-tallet. Ved å samle alle relaterte metoder og datatyper om én ting/gjenstand (såkalte adferd og egenskap) i *objekter* kunne disse omtales som en enhetlig entitet (Maus n.d.). Forskjellige objekter kunne relateres gjennom arv eller pekere. Arvingskonseptet ble tatt i bruk for å dele egenskaper og adferd mellom objekter, mens peker brukes til å snakke om eller med andre objekt fra et annet ståsted. Egenskaper i programmeringsspråkene forut for objektorientering hjalp også utviklere med strukturering av kildekoden. Blant disse struktureringsegenskapene finnes uttrykk som forårsaker løkker, *while*- og *for*-uttrykk, og rutiner og subrutiner. Da disse egenskapene smeltet sammen med objektorientering, kunne programmene splittes opp i moduler, programstrukturene kunne holdes ryddig adskilt, og det ble lagt et grunnlag for høyere abstraksjon i programmeringsprosessen. Resultatet av dette var at det ble praktisk mulig å utvikle og redigere på større og mer komplekse programmer.

Innføringen av objektorientert programmering medførte altså ikke bare at programkoden endret utseende og struktur, men den var sågar grunnlaget for en helt ny filosofi innenfor programmeringen. Det var med andre ord klart for det første viktige paradigmeskifte innen informasjonsteknologi, både på det teoretisk og på det pragmatiske plan. Objektorientering er også en av grunnpilarene i dagens moderne informatikk.

1.2 Informasjon & Design

Informasjonsdesign

Joseph A. Goguen skriver i 1997 at *«It is said that we live in an "Age of Information", but it is an open scandal that there is no theory, nor even definition, that is both broad and precise enough to make such assertion meaningful»* (Goguen 1997). Goguen etterlyser her et konseptuelt rammeverk og teori for å snakke om informasjon. Et slikt rammeverk er også nødvendig for å unngå at informasjon, som et fundament i det postmoderne samfunn, ikke skal bli etterlatt som premoderne og skjult. Informasjonsdesign (med stor 'I') er nettopp et akademisk fagfelt som tar for seg oppgavene rundt å modernisere informasjonsprosessene; et fagfelt som forsøker å gi løsninger og svar på spørsmålene som ble presentert i første avsnitt.

Det er dog verdt å merke seg er at *kvantitet* ikke (nødvendigvis) er ekvivalent med *kvalitet* hva informasjon angår (Eriksen 2001). En stor informasjonsmengde vil ofte både være uoversiktlig og den kan drukne publikumsmassen. Samtidig kan kvantitative egenskaper ved informasjonen, eksempelvis i form av multiple, differensierte fremstillinger (Karabeg 2001*b*) eller i form av utviddet kunnskap/visdoms nivå, også øke informasjonens beskaffenhet.

Struktur i Høysetet

Richard Saul Wurman argumenterer for at struktur av informasjon er løsningen for noen av de problemene vi ser i dagens informasjon: *«Organization is as important as content»* (Wurman 2001, side 10). Struktur og semantikk sidestilles således i verdi for informasjonens kvalitative egenskaper, både for de enkelte instanser (dokumenter, hjemmesider, aviser), som for den samlede informasjonsmasse (Internett). Strukturering blir altså et kjerneområde.

Det er derfor grunnleggende for denne oppgaven å definere *informasjonsdesign* som:

Definisjon 1.2.1. ***Informasjonsdesign*** *defineres her til å være strukturering av informasjon på en ny og innovativ måte.*

Strukturering av informasjon angripes altså på en måte *alternativt* til de tradisjonelle bindingene. Definisjonen er således en spesialisering av Karabegs definisjon av design: *«Design is defined as the alternative to tradition»* (Karabeg 2002), som Polyskopisk Modellerings Metode[1] er ba-

[1] Se 1.2.2

sert rundt. Spørsmålet ligger så i *hvordan* å strukturere informasjonen? Hvilke kriterier skal ligge til grunn for en slik struktur? Hvilke egenskaper skal strukturen ha? Hvordan den skal "se" ut? Dette er alle spørsmål som vil bli diskutert gjennom denne oppgaven.

1.2.1 Objektorientering og Informasjonsdesign

På den éne siden av *in medias res*: Informasjonsdesign (som fagfelt) befinner seg som nevnt i en tilsvarende situasjon som systemprogrammeringen befant seg i på 50-60—tallet. Programmererne fant på den tid løsning på sine problemer i objektorientering. En naturlig løsning på problemene informasjonsdesign støter på, vil derfor være å forsøke å foreta noen av de samme grep på informasjon, som objektorientering gjorde på systemprogrammering. Hensikten er å temme informasjonsstrukturene til håndgripelige bolker.

1.2.2 Polyskopisk Modellering og Informasjonsdesign

På den andre siden: Informasjonsvitenskap, og Informasjonsdesign i sær, er unge og lite etablerte fagområder, som fremdeles ikke er blitt gjenstand for større utprøvinger. Rammeverket og teoriene Goguen etterlyser, er under utvikling, men har ennå ikke fått skikkelig fotfeste. Det er derfor både spennende og nødvendig å ta for seg noe av de eksisterende teorier og metodologier for å sette noe av dette ut i live; å bedrive litt anvendt vitenskap.

Et slikt konseptuelt rammeverk er Dino Karabegs arbeid med Polyskopisk Modellerings Metode (Karabeg 2001*b*, Karabeg 2001*a*). I denne oppgaven er Polyskopisk Modellering lagt til grunn for de påstander og valg av løsninger som er gjort. Informasjon foreslås her presenteret gjennom multiple skop eller synsvinkler. Disse skopene er underlagt en hierarkisk struktur hvor det skilles mellom informasjon av forskjellige nivå.

En utfordring for arbeidet som er beskrevet i denne oppgaven vil dermed være å smelte sammen Polyskopisk Modellering med objektorientert programmering.

1.2.3 Polyskopisk–Objektorienterte Dokument

Denne oppgaven tar altså for seg et forsøk på å følge sporene til utviklerne bak de objektorienterte programmeringsspråkene, og å føre en analogi fra objektorientert programmering til informasjonsdesign. Resulta-

tet av denne analogien skal bli et objektorientert dokumentspråk[2] som bygger på Polyskopisk Modellering. Dette språket vil avgrense, modulere og strukturere informasjon på en ikkelineær, hierarkisk måte. Adskilt fra presentasjons og fremstillingsmåte, skal språket kunne tilby brukere på begge plan, det vil si både konsument og komponist[3], en ny og organisert måte å behandle informasjon som vektlegger tilgjengelighet, oversikt og helhetlig perspektiv.

1.3 Avklaring av Dokument–begrepet

I oppgaven er bruken og forståelsen av begrepet *dokument* tilknyttet en tosidig fortolkning som krever forklaring. Det skulle for de fleste være kjent at det finnes to separate betydninger av begrepet *bok*. En betydning av begrepet 'bok' vil være den fysiske boken, det eksemplaret, trykket og innbunnet, som kan plasseres i bokhylla hjemme. På den andre siden vil en konseptuell 'bok' være de samlinger av ord og tankegods, som satt ned på papir, utgjør dette verk — eksempelvis George Orwells *"1984"*. På samme måte kan det skilles mellom det *konseptuelle* dokumentet på den ene siden , og det *fysiske* dokument på den andre (herunder de ark/utskrifter, men også datafiler og annet digitalisert materiale som representerer dette konseptuelle dokumentet). Et eksempel på denne tvetydige bruken av begrepet finner vi i utrykket *dokumentsentriske dokumenter*, som blir introdusert i kapittel 3.1.4. Første forekomst benyttes her om det konseptuelle innhold mens, andre forekomst henviser til de "fysiske" **XML** (Extencible Markup Language (XML n.d.)) filer.

Digitalisering, og de senere års teknologiske utvikling har, ikke desto mindre hatt stor betydning på forståelsen og bruken av begrepet dokument. På norsk ble ordet dokument opprinnelig brukt om *«skriftlige redegjørelser*[4]*»*, men som et resultat av økt bruk av informasjonsteknologiske verktøy og en fagspesifikk tolkning av begrepet, har forståelsen også blitt vridd mot en mer synonym forståelse med ordet (data)*fil*. Uttrykket ovenfor er eksempel også på dette poenget. En direkte konsekvens av dette, igjen, er også en utvidet forståelse av begrepet dokument. Siden både taler, musikk, bilder, videoer, animasjoner og selvsagt også tradisjonelle tekst kan samples eller genereres som digitale versjoner uten informasjonstap (merkbart for det menneskelige sanser), vil det

[2]Dokumentspråk: språk eller verktøy for skriving og utforming av tekstlige dokumenter, herunder markupspråk som **HTML** og LaTeX, men også *wisiwyg*-varianter som Word

[3]Forfatteren bak dokumentet. Brukers for å presisere at denne både forfatter *og* komponerer dokumentstrukturen

[4]Forklaringen hentet fra Escolas Ordbok (Taule 1993)

kunne hevdes at *all* (konseptuel) informasjon kan representeres i dokumenter. Et fysisk dokument er altså ikke lenger ensbetydende med et sammenbundet sett med trykte papirer eller en annen form for skriftlig utredning.

Når ikke annet er spesifisert, vil uttrykket *dokument* bli brukt med følgende betydning i denne oppgaven.

Definisjon 1.3.1. *Et **Dokument** defineres som en avgrenset representasjon av abstrakte og konkrete objekter med eksplisitte eller implisitte budskap til mottakeren.*

1.4 Denne Oppgaven

1.4.1 Utfordring

Hovedutfordringen i denne oppgaven ligger altså i å finne en god måte å overføre konseptene *fra* objektorientert programmering og Polyskopisk Modellering *til* dokumentdesign. Resultatet skal bli ett nytt språk for dokumentdesign som blir kalt **Obdok**. Dette språket skal ta vare på de sentrale konseptene og hjelpe informasjonsdesignere å se på dokumentkomposisjon på en høyere og mer abstrahert måte.

1.4.2 Fremgangsmåte

Det første steget på veien til å designe et slikt objektorientert dokumentspråket vil være å "kvantifisere" informasjon og å lage en representasjon for slike kvanter. Disse kvantene er i denne oppgaven kalt *informasjonsenheter*, mens de respektive representasjonene kalles for *informasjonstyper*. Den konkrete representasjonen vil bli tatt hånd om av **XML**-elementer. Videre progresjon består i å søke en løsning for avgrensing og samling av relaterte informasjonstyper i entiteter, eller containere, som skal tilsvare programmeringsspråkene objekter. Disse containerne vil fungere som kontekstuelle rammer for kvantene.

Sammenheng mellom de forskjellige informasjonstypene må også beskrives, både på et abstrakt og på et konkret plan. Den konkrete sammenhengen må deretter defineres og finnes en representasjon for i **Obdok**. Videre må det bygges et strukturelt rammeverk for containerne og hvordan disse skal kunne (innbyrdes) relateres. Til slutt må det utarbeides en syntaks som kan beskrive de ulike representasjonene; både for de konkrete representasjonene så vel som for de strukturelle komposisjonene.

1.4.3 Hjelpemidler

For å lykkes i å designe et dokumentspråk som innehar disse egenskapene, er **XML** (XML n.d.) et meget velegnet verktøy. **XML** tillater for det første å definere et eget sett med elementer og en egen struktur — språket er et metaelement- og -struktureringsspråk. For det annet er **XML** velegnet da språket skiller mellom logisk struktur og fysisk presentasjon. Dermed kan fokuset konsentreres rundt utarbeiding av dokumentet og design av informasjonen *uavhengig* av hvordan dette blir seende ut. Presentasjon vil senere bli behandlet etter regler gitt i eksterne verktøy, eller språk, om man vil. Eksempel på slike er **XSLT** (Extensible Stylesheet Language Transformation (XTR n.d.)), og **CSS** (Cascading Style Sheets (CSS n.d.)). **XML** vil således fungere som det basale byggeelement for **Obdok**.

XML alene er riktignok et alt for generelt språk til kunne realisere et objektorientert språk som innehar de egenskapene beskrevet over. Den strukturelle rammen rundt **XML**-representasjonen må altså beskrives i et annet *miljø*. Dette miljøet vil ta for se de respektive informasjonstypene og deres innbyrdes relasjoner etter et eget mønster, eller en syntaks. Denne syntaksen beskriver således **Obdok** som et objektorientert dokumentspråk.

Kapittel 2

Teori

Under utarbeidelsen av **Obdok** har én metododologi og ett eksisterende paradigme stått frem som sentrale grunnpilarer. Polyskopisk Modellerings Metode er utviklet av Dino Karabeg ved Instituttet for Informatikk, Universitetet i Oslo, og er beskrevet avsnitt 2.1 & 2.2. Objektorientert programmering er også utviklet med betydelige bidrag fra norsk forskning, ved Norsk Regnesentrals Simula fra 1969. En nærmere beskrivelse av begreper og konsepter for dette, vil bli beskrevet i avsnitt 2.3

2.1 Polyskopisk Modellerings Metodologi

Polyskopisk Modellerings Metodologi er en ung metodologi som tar sikte på å løse noen av problemene rundt foredling og spredning av informasjon som ble beskrevet i introduksjonen. Grunnlaget for metodologien er lagt i åtte postulater formet som et hierarki. Fire av disse postulatene er prinsipper, de fire siste er kriterier. Prinsippene beskriver epistemologien og hvilken tilnærming informasjonsdesigneren skal benytte. Kriteriene bestemmer hvilke egenskaper som er ønskelig i informasjonen. Metodologien tar også for seg og definerer bestemte begreper, den beskriver en grunnleggende terminologi.

2.1.1 Definisjon

Polyskopisk Modellering er definert som *«information designed by scope design»* (Karabeg 2001*b*). Informasjonsdesign ønsker å strukturere informasjon slik at den lettere blir tilgjengelig for informasjonsmottaker. Begrepet *informasjonsdesign* er derfor ensbetydende med å designe, eller utforme informasjon (eng: in-formation). Polyskopisk Modellering defi-

neres gjennom et sett med regler som formaliserer og strukturerer denne formingen. Deriblant åtte *postulat* som definerer informasjon.

Metodologien er i tillegg modellert med tanke på å gi kreativiteten spillerom; dét er en av hjørnesteinene. Teknikker og metoder til bruk i informasjonsutformingen er også foreslått. Blant disse finnes ideogrammer, metaforer, bevisføring og dialoger. Felles for disse er at de alle er basert på definisjonen i Polyskopisk Modellering Metodologi.

2.2 Polyskopisk Informasjon

Forestil deg en fjellklatrer som har besteget et stort fjell. Fra toppen av fjellet har han god oversikt over landskapet under. Han kan snu seg i alle himmelretninger og se seg om, og han vil se langt utover landskapet nedenfor. Hva er det så fjellklatreren ser? Jo, han ser formasjonene av det nedenforliggende landskapet. Han ser andre fjell og daler som snirkler seg inn mellom disse, han ser heier og høyder, kanskje en kystlinje, en innsjø og et åkerlandskap. Hvis vår fjellklatrer så beveger seg nedover den ene av fjellsidene vil han oppdage et stadig mer og mer detaljert bilde av hva han har rundt seg. I det som tidligere var andre fjell, vil han nå se nye, mindre formasjoner som lokale høydedrag og åsrygger. Nede i dalen vil han kanskje se små hauger og en elv, og ute ved kysten vil han kanskje se et tettsted. Jo lenger ned vår venn fjellklatreren går fra fjellet, vil han få bedre og bedre detaljkunnskaper om det stedet han går, helt til han eventuelt kommer helt ned til den elvebredden han så, hvor han finner små steiner, små fisker og sivplanter.

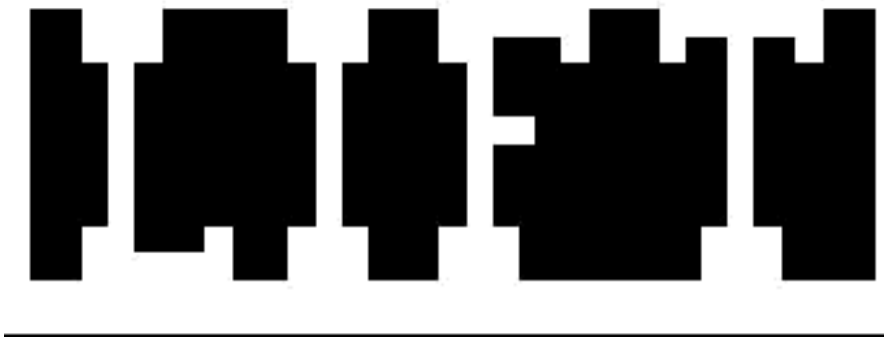
— Metaforen er lånt av D. Karabeg

2.2.1 Skop

Karabegs metafor beskriver polyskopisk informasjon som et modulbasert hierarki. Dette i motsetning til det flate og homogene tradisjonelle informasjonsbildet. Informasjonsstrukturen er tuftet på multiple, koherente *skop* (eng: scope), hvor hvert skop danner et enhetlig bilde, eller en synsvinkel av informasjonen. Synsvinkelen avspeiles leseren gjennom det skop denne har valgt. Hierarkiet strukturerer skopene i nivåer etter deres detaljrikdom. Presisformulerte og formalistiske fremstillinger presenteres således gjennom lavnivåskop, mens grovmasket informasjon legges frem i høynivå skop.

2.2.2 Perspektiv

Tilegning av skriftlig, billedlig og annen trykt informasjon, kan sidestilles med den måten mennesker tilegner seg informasjon om (andre) fysiske tingester (objekter) i naturen på. Inngående kunnskap om slike oppnås ikke utelukkende ved å studere dets detaljer og fineste kontur. Objektet må også beskues på avstand, og i fra forskjellige vinkler. Vi har alle snudd og vendt, og stilt oss på forskjellige sider, og i forskjellig avstand til en skulptur, for å erkjenne, eller oppfatte *hele* objektet som beskues. Et norsk ordtak som kan være et godt eksempel på dette er utrykket *«å ikke se skogen for bare trær»*. Figur 2.1 visualiserer det samme poeng.

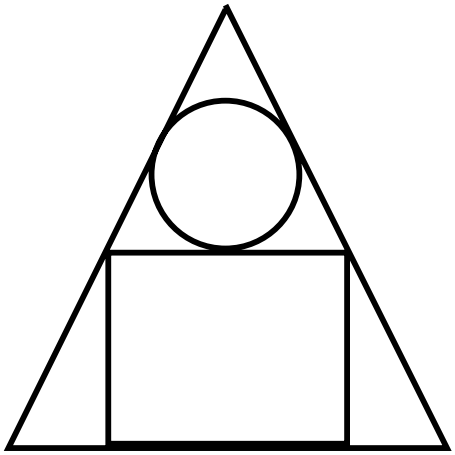


Figur 2.1: Perspektiv: er dette bare klosser eller også et ord?

Fremstilling av polyskopisk informasjonen bygger på et tilsvarende mønster som erverving av kunnskap om fysiske objekter, beskrevet ovenfor. Også slik informasjon må belyses fra forskjellige vinkler og avstander om den skal gi informasjonssøkeren et helhetlig og fullstendig bilde, eller *perspektiv*, av saken.

> «By separating the broad, wholistic views from the detailed ones, polyscopic information clearly depicts of the whole and displays the role and the importance of each detail. The details do not obscure the view of the whole (the perspective).» (Karabeg 2001*b*)

Dette skal gjøre informasjonsmassen letter tilgjenglig, på en mer effektiv måte.



Figur 2.2: Det polyskopiske hierarki

2.2.3 Informasjon

Ideogrammet i figur 2.2 viser en trekant fylt med en sirkel og en firkant. Til sammen utgjør ringen og firkantene en ***i***, et akronym som står for *informasjon*. Ideogrammet representerer ideen om hvordan polyskopisk informasjonen skal designes. Hvordan høy- og lavnivå skop skal fremstilles og hvordan disse skal bygge opp hierarkiet. Ideogrammet belyser polyskopisk informasjon ved selv å være polyskopisk modellert.

Sirkelen Øverst i hierarkiet finnes informasjon av høy abstraksjon. Formålet er å forsyne informasjonssøkeren helhetlig oversikt og perspektiv. Den *lukkede* sirkelen og den *runde* formen symboliserer helhet og oversiktlig forståelse. Informasjonen på dette nivået skal ikke preges av detaljering og utbroderte forklaringer — detaljer vil være skavet bort. En metafor for dette kan være en skulptør som skisserer det første utkastet til en figur eller tegning. Virkemidler fra kunsten hentes da også inn for å hjelpe informatøren med å forme informasjonen slik at perspektivene bevares. Slike virkemidler kan for eksempel være ideogrammer eller metaforer. Høynivå informasjon skal fungere som veiledning for fjellklatreren på vei ned fra "informasjonsfjellet" (Karabeg 2000*a*, Karabeg 2001*a*).

Legg merke til at kunst og vitenskap er ikke tenkt blandet til et ustrukturert sammensurium, men at de sammen skal bidra til å formidle informasjon på en korrekt, forståelig og oversiktlig måte.

Firkanten Firkanten fyller ut triangelet i bunn og står stødig på dets grunnlinje som et fundament for sirkelen over. Informasjon på dette lave nivået skal forklare dypere, begrunne, underbygge og gå i detalj til de høynivå fremstillinger rundt skissen. Informasjonen opptrer som vitenskaplig, konkret og utfordrende. Synsvinkelen på dette lavnivået skal være objektiv, presis, detaljrik og bygge på en vitenskaplig tone. Språket blir gjerne preget av å være teknisk og fagspesifikt.

2.2.4 Summa Summarum

Summen av disse tre velkjente geometriske figurer utgjør et ideogram som forteller noe om fundamentene i polyskopisk informasjon. Hvis informasjon blir gitt etter disse prinsipper vil den *helhetlige* forståelse øke (Karabeg 1999, Karabeg 2000*a*). Perspektivene skal konkretisere det helhetlige og hjelpe til å belyse den fulle *sannheten* og således skille den fra de ufullstendige fragmentene.

2.3 Objektorientert Programmering

Hva er det som gjør objektorientert programmering så fordelaktig i forhold til andre programmeringsparadigmer? Hvilke egenskaper har Simula, Java, C++ og andre objektorienterte språk som gjør de så hensiktsmessige å benytte — spesielt til store programsystemer — og hvordan hjelper disse systemutvikleren til å beholde oversikt over både hendelsesflyt og datastruktur. Hva har disse språkene i seg som muliggjør gjenbruk, reforståelse og videreutvikling, og som sådan øker et systemets levetid.

Ved å studere de fordelaktige egenskapene ved objektorientert programmeringsspråkene, klargjøres også de egenskapene som er ønskelige å la nedarve i syntaks og struktur for objektorienterte dokumentspråk. I det følgende presenteres derfor grunnprinsippene i de objektorienterte språkene og hvilke byggesteiner disse er tuftet på. Kapittel 4 tar så for seg hvordan et objektorientert dokumentspråk kan bygges opp. Her beskrives byggesteiner, grunnprinsipper og syntaks for **Obdok**.

2.3.1 Byggesteiner i Objektorienterte Språk

Datatypene

Å programmere handler om manipulering av data. Disse data blir re-

presentert i en datamaskin ved hjelp av forskjellige komposisjoner av bits og bytes. Fortolkningen av slike, om en prosessor er Big-Endian eller Litle-Endian, og hvor mange bit og bytes som brukes i lagringsenhetene et cetera, er av liten interesse i denne sammenheng. Direktemanipulering av slike byggesteiner eller lavnivå assembly-programmering er også utenfor interessefeltet. Derimot vil fokuset ligge på et noe høyere nivå: på det nivået som er synlig for en moderne programmerer. Fokuset vil naturligvis ligge på å beskrive grunntrekkene ved objektorienterte programmeringsspråk, som Java, C++ og Simula. Dette vil også omfatte egenskaper som overlapper med andre, generelle hendelsesorienterte funksjonsspråk, som C, Fortran og Pascal. Felles for alle disse er at de klassifiserer inn under en gruppe som kalles *van Neuman*-språk, *imperative* språk eller uttrykksbaserte språk.

Utgangspunktet for slik datamanipuleringen finnes i et begrenset utvalg av lagringsvariable, kalt *datatyper*. Disse datatypene gruppers gjerne i to klasser:

Definisjon 2.3.1. ***Primitive datatyper*** *er de datatyper som* ikke *er bygget opp av andre datatyper*

Definisjon 2.3.2. ***Komplekse datatyper*** *er de datatypene som* er *bygget opp av en eller flere andre datatyper*

Blant de primitive datatypene er heltallsvariable, boolske verdier, karaktertegn (eng: character) og desimaltall. Slike datatyper deklareres og manipuleres gjennom forskjellige *variable* av datatypens sort. Typisk for disse variablene et er de lagrer én dataenhet (måldata) per variabel. Eksempelvis vil en integer variabel kunne lagre maksimalt *én* persons alder i antall hele år, eller en boolean ville kunne lagre *én* logisk sannhetsverdi.

Eksempler på de komplekse datatyper kan være rekker (eng: array) i én eller flere dimensjoner, *strukter*, tekststrenger (som jo egentlig er rekker av karaktertegn), objekter osv. Komplekse datatyper *kan* være utelukkende bygget opp av en primitiv datatype. Et eksempel vil kunne være meteorologiske data, si nedbør siste 30 døgn i antall millimeter. Denne kan representeres med en rekke av heltall (integer-array). Komplekse datatyper kan derimot også være bygges opp av *forskjellige* variable av både primitive og komplekse datatyper. Et objekt som representerer et sted (plass i landet) kan eksempelvis være bestående av meteorologirekken beskrevet ovenfor, en tekststreng som benevner stedets navn, samt to desimalvariable som tar vare på stedets bredde- og lengdegradsverdier.

Objekter

Ideen bak innføringen av objekter var å kunne samle relaterte data om en entitet innenfor én container, samt å kunne gi hver slik container en rekke operasjoner, eller en viss funksjonalitet. Dette er operasjoner som håndterer objektets interne data eller objektet i seg selv.

Terminologien definerer et objekt til å være en *instans* av en *klasse*. En klasses egenskaper, det vil se de datatyper og den oppførselen denne skal ha, defineres i klassens deklarasjon. Denne klassen kan altså være mal til flere *instanser* (objekter) med de samme eller med lignende egenskaper og oppførsel.

Pekere

I tillegg til at programmeringsspråkene tilbyr direkte aksess til datalagringstypene gjennom opprettelse av variable, kan også *pekere* eller dataminnereferanser benyttes. Særlig nyttig er dette for manipulering på objekter. Ved å opprette en peker — egentlig en variabel inneholdende minneadressen til det objektet eller den datatypen denne "peker" på — kan fjerne objekter manipuleres i sin helhet, deres interne variabler kan nås, og objektenes funksjoner kan kalles. Pekere er også nyttig til oppretting av strukturer som lister, trær og grafer. Hvert objekt utstyres da med pekere, hvis oppgave er å knytte de forskjellige objektene sammen. Eksempelvis vil objektene i en (enveis) liste peke på neste objekt i listen, mens objektene (CS: node) i trær peker på barnenodene.

2.3.2 Funksjoner og Funksjonalitet

Utvikling av programsystemer følger gjerne en kravsliste for å sikre at alle forlangende blir oppfylt. Disse kravene danner utgangspunktet for programmets *funksjonalitet*, også kalt programmets intensjon. I "Object-oriented Analysis & Design" (Mathiassen, Munk-Madsen, Nielsen & Stage 2000) sidestilles krav og funksjonalitet og omtales som *«an abstract property of the system»*. Funksjonalitet defineres til å være:

Definisjon 2.3.3. ***Funksjonalitet:*** *En fasilitet for å lage en brukervennlig modell.*

Den reelle funksjonalitet oppstår som et resultat av de handlinger og kommandoer et program tillates (etter koden) å utføre på de underliggende datatyper og -strukturer. Det kan også sies at et programs samlede funksjonalitet summeres opp av alle slike kommandoeksekvering. Hver

eneste lille enkelthandling, eksempelvis en manipulasjon av en datavariabel eller lesing av en referanseverdi, er med på å utgjøre den samlede funksjonaliteten. I laverenivås programmeringsspråk, som binær- og assemblykode, finnes ingen måte å strukturere disse enkelthandlingene. Koden blir derfor langstrakt (endimensjonal), rotete og uhåndgripelig å lese for det menneskelige øye.

Et vellykket forsøk på å rydde opp i slik rotete kode, gikk ut på å samle et sett enkelthandlinger som til sammen ville beskrive en felles operasjon. En slik samling hendelser ble gitt navnet *funksjon* eller *prosedyre*. En funksjon består av et navn, eventuelle inn- og returparametere, samt den samling av kommandoer/kodelinjer som utgjør operasjonene. En slik funksjon kan bli kalt i programmet som en rutine, mens subrutiner, som utfører deler av en slik funksjon, kan skilles ut som separate rutiner. Dette bidraget medførte en avgrensning og gruppering av de forskjellige funksjonalitetene til et program, med et resultat av bedre oversikt over hendelsesflyten i programmets eksekvering. Gruppering og navngiving av hendelsesflyt forenklet også samtale om disse operasjonene.

Inndelingen av rutiner og subrutiner gir programstrukturen en hierarkisk karakter. Lesing og tilegning av oversikt over programkoden forenkles som følge av dette, sett i forhold til den én-dimensjonelle (lineære) assamblykoden. Modulinndeling av programkoden medfører også modulering av selve funksjonaliteten. Denne funksjonalitetsinndelingen inntar den samme hierarkiske struktur.

2.3.3 Arv

Objektenes evne til å arve egenskaper fra andre objekt(er) har også vært en viktig medvirkende faktor for å oppnå kortere, mer lettlest og ikke minst mer elegant kode. Hovedprinsippet går ut på at *sub*-klasser arver egenskaper fra *super*-klasser. I den virkelige verden finnes mange gjenstander/ting som deler flere felles trekk med hverandre, men som har enkelte individuelle forskjeller. Slike konstellasjoner representeres gjerne i objektorienterte språk ved å definere den generelle klassen først. Her deklareres alle egenskaper felles for klassesystemet. Deretter defineres underklasser av denne. Disse arver superklassens egenskaper, men spesialiseres etter deres særegne behov. Et eksempel kan være gruppering av forskjellige kjøretøy. Superklasse ville kunne være 'kjøretøy', mens subklassene kunne være 'lastebil', 'personbil' og 'motorsykkel'. Lastebilklassen kan igjen være superklassene for 'vogntog' og 'trailer', personbil for 'varebil', 'cabriolet' og 'sedan' , og motorsykkel kan ha subklassene 'touring', 'racing' og 'chopper'. Det hele dreier seg altså om klassifisering av gjenstander — hvilke gjenstand som skal i hvilke bås,

og hvilke særegenskaper objektene i hver bås skal få.

På tilsvarende måte som objekter arver egenskaper, arver de også funksjonalitet gjennom de funksjonene som er deklarert i superklassen. Også disse funksjonene kan redefineres for "skreddersøm" i subklassene, gjennom såkalte *virtuelle* prosedyrer. Disse virtuelle prosedyrer kan også deklareres tomme og eventuelt instanseres i subklassene.

2.3.4 Komplekse Strukturer — Abstrakte Datatyper

Under arbeid med problemløsning er det ofte ønskelig å skave av det opprinnelige problemet og forme en mest mulig generell løsning, slik at denne kan gjenbrukes på andre, lignende problemer. En slik prosess kalles *abstraksjon*, og løsningen søker å heve seg over problematikken rundt valget av datalagringstyper og implementasjon. Mer spesifikt kan det sies:

Definisjon 2.3.4. *En **Abstrakt Datatype** (ADT) er et implementasjonsuavhengig grensesnitt med en presist spesifisert oppførsel — d.e. det sett av funksjoner som utgjør ADT'en.*

Definisjon 2.3.5. *Et **Grensesnitt** er en felles kommunikasjonsplattform mellom entiteter.*

Mens de konkrete datatypene fokuserer på et verdibegrep, vil ADT'ene fokusere på manipulasjonsmulighetene — eller altså grensesnittet. Når all kommunikasjon til ADT'en går via aksessprosedyrer isteden for direkte på datarepresentasjonene, kan det øvrige program se bort i fra hvordan denne ADT'en er implementert. En slik *black-boxing* utvidder således modulinndelingen av programmet:

> «Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of how the set is implemented. This can be viewed as an extension of modulardesign.» (Weiss 1995).

Forskjellige implementasjoner av en ADT baserer altså sitt grensesnitt rundt ulike strukturelle og representasjonsmessige løsninger. En stakk (LiFo–kø) kan eksempelvis både bygges opp av en array óg av en lenket objektliste, så lenge de begge tilbyr samme funksjonalitet. Datastrukturene som ligger til grunn i implementasjoner av slike ADT'er er sjelden så simplistiske som i listetilfellet. Mange typer problemer innen informatikken (CS) ender gjerne opp i mer komplekse formasjoner som tre- og grafstrukturer.

“

ADT og Arbeid med Obdok

Trestrukturer blir også flere steder benyttet som modeller eller basale grunnstrukturer i arbeidet med **Obdok**. Blant annet finner vi igjen trestrukturen i det polyskopiske hierarki (2.1) og i **DOM**-representasjonen av **XML**-dokumenter (kapittel 3.4 på side 29). En trestrukturrepresentasjon av **XML**-dokumenter kan, som det senere vises, være en nyttig innfallsvinkel spesielt ved manipulasjon av og søk i slike dokument. Trær blir også nyttet som modell for å beskrive oppbygningen av ulike abstrakte informasjonstyper i kapittel 4.3.3 på side 45.

Avsnittet om objektorientert programmering 2.3 støtter seg spesielt på to informasjonskilder, i tillegg til oppslagsverk og internettside. Det første av disse er *"Objektorientering og systemutvikling — en kort innføring"* (Maus n.d.) . Den andre kilden er *"Data Struktures and Algoritm Analysis"*(Weiss 1995).

Kapittel 3

Verktøy & Teknologier

Utarbeidelsen av **Obdok** er sentrert rund Extencible Markup Language (XML n.d.) fra World Wide Web Consortium, samt øvrige **XML**-relaterte verktøy. Utover dette finnes en rekke mer eller mindre sentrale verktøy som er blitt benyttet under utforming og skriving av oppgaven. **XML**-orienterte teknologier og programmeringsspråket Python blir beskrevet i dette kapittelet, mens for beskrivelse støtteverktøy, plattform og nettlesere, deriblant Emacs, Linux, Mozilla, LaTeX og Acroread, henvises til andre, åpne informasjonskilder for nærmere beskrivelse.

3.1 Valg av verktøy

Det er flere grunner til å velge **XML** som hovedverktøy til å utforme dette objektorienterte språket for dokumentkomposisjon. Deriblant er verktøyet nytt og må således utforskes for å finne muligheter og begrensninger ved det. At **XML** er et nytt språk med nye muligheter er også et utgangspunkt for denne oppgaven. Dette blir utdypet i det inneværende kapittel.

3.1.1 Dokumentdesign

Den første og mest iøynefallende grunnen til å velge **XML** som utviklingsverktøy, er at verktøyet tillater brukeren å definere sin egen dokumentstruktur samt å spesifisere dennes innhold; det vil si å angi hvilke elementer strukturen er bygget opp av. Da det er nettopp *dette* som er essensen i denne oppgaven, lar det seg forstå at teknologien ikke bare en grunnpilar i oppgaven, men også et utgangspunkt for den. **XML** var med andre ord ikke utelukkende et verktøy som ble valgt for å bistå på

veien mot et mål. Tvert imot er oppgavens utforming og karakter like så mye et resultat av at dette nye verktøyet er blitt tilgjengelige.

På den ene siden fører nye verktøy med seg et behov for læring og forståelse. Der må legges et grunnlag for fornuftig bruk av verktøyene, samt at disses bruksområder og begrensninger må kartlegges. *Actor Network Theory (ANT)* bruker begrepet inskripsjon (eng: inscription) om dette — *«the way technical artifacts embody patterns of use»* (Ole Hanseth 1998). Inskripsjon avdekkes gjennom bruk av teknologien. Den naturlige forlengelse av dette er spørsmålet om det er bruker som styrer eller blir styrt av teknologien (sosiologisk eller teknologisk determinisme), er også viktig og interessant. Dessverre faller dette utenfor denne oppgavens rammer.

Sett fra én synsvinkel, inviterer nye **XML**-relaterte verktøy Informasjonsdesign, som faggruppe, til virkelig å kunne frigjøre seg fra tradisjonsbindene tøyler. Samtidig muliggjør redskapene også materialisering og iverksettelse av nye ideer og designkonsept. Begge disse anliggende er denne oppgaven et eksempel på.

XML er elementbasert dokumentspråk hvor komponisten er fri til å bestemme elementenes navn, sammensetning, semantiske betydning og visuelle utforming. Dette gjør at språket evner å være både fleksibelt og modulerbart. Disse egenskapene er det i særskilthet som skiller språket fra forgjengeren **HTML** (HyperText Markup Language (HTM n.d.)); et språk som må sies å være låst av:

- et begrenset utvalg av elementer
- en begrenset handlefrihet innenfor disse elementenes innbyrdes sammensetning (det vil si dokumentets struktur)
- elementenes semantiske og visuelle betydning var forutbestemt[1].

3.1.2 Stilsett

Ved å la et eget stilsett (eng: stylesheet) beskrive den visuelle fremstillingen, kan informasjonsdesigneren konsentrere fokuset på selve **XML**-dokumentet og dets interne struktur og innhold. Denne arbeidsoppdelingen bidrar også til frigjøring av informasjonsdesignerens fokus: egendefinerte strukturer på tvers av tradisjonelle og kulturelle bindinger lar seg dermed bygge. Det er også et fortrinn at logikk og presentasjon er adskilt, for slik kan et dokument legges frem på utallige forskjellige vis,

[1] Visuell fremstilling er ikke lenger så strekt knyttet med **HTML 4.0**, hvor **CSS** skulle ta hånd om de visuelle delene ved nettsidene

etter behov, og med forskjellige stilsett/løsninger. De to mest anvendte stilsettene er **CSS** (Cascading Style Sheets (CSS n.d.)) og **XSLT** (Extensible Stylesheet Language Transformation (XTR n.d.)). Sistnevnte er beskrevet i avsnitt 3.3

3.1.3 Semantiske Tagger

En annen fordel ved å velge et språk som **XML** for design av informasjon, er språkets innebygde evne til å tilby både humane og inhumane brukere metainformasjon. Ikke bare er det mulig å komponere en hver brukerspesifisert tagg for innkapsling av metainformasjon, i motsetning til **HTML**s predefinerte *<meta>*-tagg. Viktigere er det at hvert enkelt elements navn (i seg selv) kan være informasjonsbærende. *<p>*-taggen fra **HTML** er i seg selv ikke veldig meningsfull for den menneskelige leser. I **XML**, derimot, kan et element *pris* defineres med et tilhørende attributt *valuta* (se mer om syntaks i **XML** 3.2.1). Om et menneske (fortrinnsvis en av germanskspråklig bakgrunn) skulle lese dette dokumentet og komme over konstellasjonen:

```
<pris valuta="kroner">499</pris>
```

vil denne personen intuitivt forstå den semantiske betydningen av elementet, selv uten kunnskaper om **XML** eller om temaet dokumentet omhandler. Hvis dokumentet derimot leses av en datamaskin, må denne på forhånd være programmert til å tolke betydningen av slike utsagn. Så lenge en prosedyre *"vet"* at elementets innhold (499) er et tall som angir en pris gitt i *valuta* sin verdi (kroner), kan denne prosedyren gjøre utregninger med dette tallet. Slike utregninger kan være å beregne beløpet til andre valutaer (j.fr gjeldene kurs), eller å summere opp alle objektene i en kundes handlekurv.

3.1.4 Datasentriske og Dokumentsentriske Dokumenter

Det er verdt å legge merke til at **XML** også er velegnet til andre typer oppgaver enn utelukkende å lage hjemmesider og støtte til nettapplikasjoner. Typisk skilles det mellom to typer **XML**-dokumenter, *datasentriske* (Data-Centric) *dokumentsentriske* (Document-Centric) dokumenter. Den førstnevnte typen dokumenter er gjerne laget for å bli benyttet av andre systemer, være seg **XML**-databaser, elektronisk datautveksling (EDI), e-Business applikasjoner eller lignende. Slike dokumenter er gjerne preget av tabellstruktur eller repetitive mønstre. Dokumentsentriske dokumenter er på den annen side laget med henblikk på at mennesker skal benytte seg av informasjonen; da gjerne etter at det tekstlige innhold er blitt

transformert og/eller tillagt stil gjennom et stilsett (**CSS** eller **XSL(T)**). Slike dokumenter har mindre grad av gjentagende struktur, men er gjerne heller ikke så komplekse som datasentriske dokumenter.

3.1.5 Avrunding

Til nå er kun de spesielle egenskapene ved **XML** presentert; de egenskaper som skiller språket i større eller mindre grad fra tradisjonelle dokumentspråk. Dette kan være språk **SGML, HTML, LaTeX**, eller *wisiwyg*-verktøy som **Word** og **FrontPage**. Språk/verktøy, som i kraft av at sin syntaks, tvinger bruker til skrive innefor en begrenset ramme, hva struktur, utseende og komposisjon gjelder. Språkene og verktøyene stadfester således tradisjonen, samtidig som de undertrykker mangfold. **XML** er derimot et språk som åpner for å lage *forskjellige* dokumenter med *forskjellige* dokumentstrukturer. Hvordan språket realiserer dette, og hvordan komponisten skal sy sammen **XML**-dokumenter, beskrives i forestående avsnitt gjennom forklaring av **XML**-syntaks og dokumentkomposisjon.

3.2 Extensible Markup Language 1.0

XML er et metataggespråk spesifisert av World Wide Web Consortium (XML n.d.). Språket lar bruker selv lage sine egne elementer med kontekstsensitive navn, bestemme attributtmengde samt å definere elementenes innbyrdes og interne sammensetning. På denne måten kan også hele dokumentets strukturelle oppbygging spesialkomponeres. **XML**-dokumenter er bygget opp av elementer, hvor i de databærende enhetene enten representeres som tekst (Parsed Character Data — *#PCDATA*) eller som attributtverdier. Nedenfor er det på en mer inngående måte vist hvordan dokumentstrukturer bygges opp ved hjelp av elementer, #PCDATA og attributter, og hvorledes dette mønstret på forhånd kan spesifiseres i en dokumenttype-definisjon (**XML**: Document Type Definition – **DTD**).

3.2.1 Elementer og Attributter

XML er et subsett av **SGML** og supersett til **HTML**. Syntaktisk er da også disse språkene nært beslektet, men den store forskjellen ligger altså i at **XML** er et språk for å opprette *egne* tagger. Basisbyggesteinene er *elementene*. All semantisk informasjon et dokument innehar er (på et eller

annet vis) lagret i disse, være seg som innhold av elementene, elementets navn, eller elementets *attributter*. Der finnes sogar *tomme* elementer; *
*- og *<hr />*-taggene i **HTML** er eksempler på slike. Mer vanlig forekommer dog tomme tagger med attributter. Et eksempel på dette kan være **HTML**s *image*-element. Mens generelle elementer bygges opp av en starttagg, innholdet, og en sluttagg, inneholder tomme elementer kun starttagg med eventuelle attributter. Legg merke til at tomme tagger i **XML** *må* avsluttes med bakslasktegnet *'/'*. Tabell 3.1 viser eksempler på konstruksjon av tagger og elementer:

```
<start_tagg>
</slutt_tagg>
<tomtagg attributt="attributt-verdi" />
<tagg>innhold</tagg>
<!- -kommentar- ->
```

Tabell 3.1: Eksempel i XML-syntaks

Siden **XML**-spesifikasjon er utformet i et engelsk språk, er også nøkkelordene og begrepsapparatet rundt språket tuftet på engelsk.

Innhold — #PCDATA Datalagring foregår altså enten som attributtverdier eller som innhold til elementer. Attributtene tar vare på data som tekststrenger (*CDATA* — Character Data) inne i elementets start- eller tomme tagg. Elementenes innhold kan enten være nye elementer — og substrukturer av sådanne — eller det kan være #PCDATA; data som ikke prosesseres annet enn ved at det legges til et eventuelt stilsett. Det er #PCDATA som utgjør hovedtyngden av den informasjonsbærende biten i et dokument.

Substrukturer Substrukturer dannes når et superelements innhold komponeres av ett eller flere subelementer, som igjen på sin side kan være bygget opp av nye subelementer. Superelementenes start- og sluttagger må omslynge sine inneholdte subelementer fullstendig; nesting av elementer er ikke tillatt. Eksemplet under viser en forbudt konstruksjon:

```
<super><sub></super></sub>
```

Deklarasjon En annen restriksjon til komposisjonen er at det bare skal finnes ett *rot*-element som ytterste element. Dokumentet skal også inneholde en *deklarasjon* på formen:

```
<?xml version="1.0" '<encoding>*' '<standalone>?' ?>
```

CDATA Utover at elementer kan ha tekstlige eller substrukturelle innhold, kan de også ha såkalte CDATA-avdelinger, og kommentarer. Et eksempel på en CDATA-avdeling kan være:

```
<![CDATA[ <hullo name="world">Hallo Verden</hullo>]]>
```

Innholdet i en CDATA-avdeling vil under prosessering ordrett bli matet ut (eng: verbatim). Alle tegn som befinner seg innenfor de innerste rammeparentesene vil bli omgått, isteden for å bli tolket som annen merket data. Motsatt vil alle tegn som befinner seg innenfor kommentartegnene (<!– – ... – –>) sløyfes under prosessering.

Valide og Velformede XML-Dokument

Ovenfor er det nevnt en rekke egenskaper ved **XML**-dokumenter, deriblant krav og restriksjoner til syntaksen. Foruten disse finnes i spesifikasjonen (XML n.d.) en rekke øvrige skranker til et korrekt **XML**-dokument. Dokumenter som følger disse krav, sies å være *velformede* **XML**-dokumenter. Et velformet **XML**-dokument har altså en korrekt **XML**-syntaks. Et velformet dokument er også *valid*, eller gyldig, om det finnes en assosiert **DTD** og dokumentet følger denne.

XML er en sluttet syntaks, noe som medfører at velformede instanser også er programmerbare. Det vil si at et **XML**-dokument kan kontrolleres automatisk, både på velformethet og validitet. Spesielt under arbeid med store dokumenter, gjerne med høy kompleksitet, vil slike automatiske validatorer være til stor hjelp. Validatorer finnes i forskjellige typer, både som onlinevarianter på Internett, eller som applikasjoner som kan kjøres på plattform.

3.2.2 Document Type Definition

En dokumenttype-definisjon (**DTD**) er en formell definisjon av en **XML**-struktur. En slik definisjon beskriver det innbyrdes forhold mellom de forskjellige elementene, og mellom elementene og deres attributter. En **DTD** bygges opp over en liste, som beskriver elementenes sammensetning og deres attributter. **XML**-elementene kan dermed sees på som materialiseringer av dokumenttype-definisjonen. Dersom denne materialiseringen følger **DTD**'en er dokumentet altså valid.

Elementdefinisjon

Et element defineres ved å bestemme dets navn og ved å liste opp dets

lovlige innhold. Definisjonen av innholdet beskriver hvilke komposisjoner av tekst og/eller substrukturer som utgjør dette elementet. Skal elementet være tomt settes dette med nøkkelordet *EMPTY*. Substrukturene kan eksempelvis bygges som strenger, sekvensielle rekker eller mer tilfeldige og randomiserte kombinasjoner. Et element kan også bestå av flere forskjellige kombinasjoner. Disse substrukturene grupperes ved hjelp av parenteser. Et hvert subelement og en hver samling kan gis pluralitet. Tabell 3.2 viser pluralitets- og sekvensskrankene i **DTD**:

,	sekvensopprasing
*	null eller flere forekomster
+	én eller flere forekomster
?	null eller én forekomst
( )	avgrensing av flere subelementer i en samling
|	logisk *'eller'*
&	logisk *'og'*

Tabell 3.2: Skranker i XML

Attributtdefinisjon

For et elements attributter settes også attributtets navn, om dette er påkrevet eller valgfritt, og om det kan gis en eventuell forhåndsbestemt (eng: default) verdi. Påkrevde attributter merkes med nøkkelordet *REQUIRED*; om attributtet er valgfritt, merkes dette som *IMPLIED*; ugjenkallelige forhåndsbestemte verdier merkes med *FIXED* pluss verdien. Brukerbestemte data kommer som tekststrenger, såkalte (*CDATA*). Attributtverdien kan også oppgis som predefinerte valg, eller grupperinger. Da med stavtegnet (eng: bar — '|') som skilletegn mellom valgmulighetene.

Assosiert DTD

Det finnes to måter å relatere et **XML**-dokument opp mot en dokumenttype-definisjon; enten internt i **XML**-dokumentets hode, eller eksternt i en egen, separat fil. Det kan også være verdt å merke seg at et **XML**-dokument kan være et velformet **XML**-dokument selv om det ikke har noen relatert **DTD**. En slik løsning blir likevel en *ad-hoc*-løsing som gjerne blir vanskelig å gjenbruke.

Tabell 3.3 gir et eksempel på en dokumenttype-definisjon.

```
<!DOCTYPE liste [
<!ELEMENT liste (album)*>
<!ATTLIST liste type (lpliste | cdliste) “cdliste”>
<!ELEMENT album (band, (produsent | plateselskap)*, utgitt)>
<!ATTLIST album tittel CDATA #REQUIRED>
<!ELEMENT band #PCDATA>
<!ELEMENT utgitt EMPTY>
<!ATTLIST utgitt år CDATA #REQUIRED»
<!ELEMENT produsent #PCDATA>
<!ELEMENT plateselskap #PCDATA>
]>
```

Tabell 3.3: DTD-eksempel

3.3 Extensible Stylesheet Language Transformation

XSLT (XTR n.d.) er et støttespråk for å transformere eksisterende **XML**-dokumenter til andre dokumenter. **XSLT** er designet for å være en del av det uferdige **XSL** (Extensible Stylesheet Language (XSL n.d.)), men er også mulig å bruke uavhengig av dette — for eksempel til transformering fra **XML** til **HTML**. **XSLT** kan også benyttes til å lage presentasjoner, legge til (forskjellig) stilsett, endre på dokumentstrukturen (særlig anvendelig for arbeid med datasentriske dokumenter), sortere trær samt mer komplekse kalkuleringer. Syntaktisk er **XSLT** et **XML**-dokument, bestemt av et så kalt *namespace*. **XSLT**-dokumentet vil således inneholde både elementer med attributter og innhold, på lik linje med andre **XML**-dokumenter.

3.3.1 Funksjonelt Språk — Templates og Regler

XSLT er et funksjonelt språk. *Template*-reglene i stilsettet bestemmer hvorledes elementene i **XML**-dokumentet skal prosesseres. Eksekveringen er dermed ikke hendelsesorientert, iterativ og sekvensiell, i motsetning til hva som er tilfellet med eksekvering av objektorienterte programkode.

Selve transformeringen av **XML**-dokumenter foregår over tre steg. Det første steget leser inn dokumentet, parserer strukturen og oppretter en **DOM**-trerepresentasjon av dette (for **DOM**, se kapittel 3.4). Dernest transformeres de enkelte nodene i trestrukturen ved at templatene i stilsettet samsvarer (eng: match) nodene og utfører sine templateregler på disse. Til slutt skrives det nye **XML**-dokumentet ut til fil eller skjerm.

I **XSLT**-stilsettet finnes såkalte templateelementer, som skal sammenfalle med de forskjellige nodene fra det parserte treet. Når en slik template

har fått et treff på en elementnode, forteller templatets regelsett hvordan elementet skal prosesseres. Der finnes egne **XSLT**-regler for både sekvensiell og rekursiv gjennomløping av elementnoders subelementnoder; egne regler for å hente ut verdier av noders innhold eller attributtverdier; og egne regler for å gjøre utvalg i, og teste på egenskaper. Trestrukturen kan også navigeres for å hente ut noder fra andre lokasjoner i treet. Denne navigeringen bygger på **XPath**-språket (Pat n.d.). **XPath** tar utgangspunkt i **DOM**-modellen (beskrives i neste avsnitt) og dets funksjoner tillater et mer nøyaktig utvalg av noder fra dokumenttreet. Navigasjonen følger både vertikale og horisontale akser, og blant de tilgjengelige aksene er *child, parent, descendant* og *preceding-sibling*. Det finnes fire typer basisfunksjoner klassifisert etter returverdiene. Disse returverdiene er, tall, nodesett, strenger, eller boolske verdier. Funksjonskallene kan være både med og uten argumenter. Eventuelle argumenter kan sendes med for å kontrollere returmengden.

3.3.2 XML Linking Language — XLink

W3C har også spesifisert en egen standard for linking av resurser (W3C: resources)[2] som er naturlig å nevne i forbindelse med **XML**. Dette språket heter **XML** Linking Language eller **XLink** (Lin n.d.). Språket benytter **XML**-syntaks, og tilbyr i tillegg til regulær *"simple-link"* — tilsvarende **HTML**s hyperlink (*<a>-taggen*) — en mer kompleks og sofistikert link, *extended-link*. Denne linktypen kan knytte sammen et utall resurser i en link. Språket muliggjør også å assosiere metadata til en link, og til å plassere linken adskilt fra de(n) linkede resurs(er).

På grunn av manglende implementasjoner av **XLink** i de verktøy og hjelpemidler som er benyttet i utarbeidelsen av denne oppgaven, baseres dokumentlinking i **Obdok** på (en omskriving av) **HTML**s *anchor-element*.

3.4 DOM-Modellen & Parsering

Én måte å fortolke **XML**-dokumenter, er gjennom å se på elementoppbygningen som en trestruktur. **DOM** (Document Object Model (DOM n.d.)) er et grensesnitt med en slik forståelse av dokumentet. Utarbeidet av W3C, er **DOM**-APIen[3] en plattform- og språknøytral modell, som tilbyr program og skript dynamisk tilgang til dokumenters innhold, struktur

[2] *«a resource is any addressable unit of information or service»* `http://www.w3.org/TR/xlink/\#N789.>>`

[3] Application Program Interface

og stil. **DOM** tillater således programmereren å lese elementer, attributter, kommentarer, #PCDATA og andre elementtyper, å manipulere på disse og deres struktur, samt å skrive og opprette nye trestrukturer.

Blant de mest sentrale grensesnittene i spesifikasjonen, finnes *Document*, *Element*, *Attr* & *NodeList*. Deres egenskaper er:

- **Document**-grensesnittet presenterer/inneholder *hele* **XML**(/**HTML**) dokumentet.
- **Element**-grensesnittet representerer et **XML**-element og begrenses av start- og sluttaggene (eventuelt tomme elementer). Innhold (for eksempel #PCDATA eller subelementer) og attributter er tilgjenglig gjennom dette grensesnittet.
- **Attr** representerer et elements attributt, med navn og verdi.
- **NodeList** er en ordnet liste av utvalgte noder.

Disse grensesnittene tillater bruker å navigere, gjøre utvalg, og omstrukturere dokumentene, på samme måte som et hvilket som helst databasespråk. **DOM**-modellen tillater derfor bruker å se på **XML**-dokumenter i databaseperspektiv. Native **XML** Databases (**NXD**), basert på **XPath**, er da også en slik **XML**-database.

DOM vs. SAX

En **DOM**-parser er altså et redskap som leser inn en **XML**-struktur og oppretter et tre av denne. Dette i motsetning til en **SAX** (Simple API for **XML** (SAX n.d.)). Dette er en hendelsesorientert parser som bryter opp en **XML**-struktur til lineære, eller sekvensielle, begivenheter. En starttagg vil således opptre som en hendelse, så vil også et attributt, og en #PCDATA-seksjon. Den store fordelen med en slik tilnærming, er at den er effektiv til søk i innhold, resursbesparende (i forhold til en **DOM**-representasjon som må lagre hele strukturen i minnet), og at den er enklere konstruert. Ulempen går på manaurerings- og manipulasjonsmulighetene, hvor trerepresentasjonen er uovertruffen.

Eksemeplet:

```
<produkt>
  <pris>15</pris>
</produkt>
```

vil i **SAX** prosesseres til:

```
start element: 'document'
start element: 'produkt'
start element: 'pris'
characters: '15'
end element: 'pris'
end element: 'produkt'
end element: 'document'
```

I sammenheng med beskrivelse av disse to parserene, er det også verdt å merke seg at selve analysen hos **DOM**-parsere nettopp foretas av en hendelsesorientert parser. Det er først og fremst i at **DOM**-parseren oppretter trestrukturen at disse parserene skiller seg fra hverandre.

DOM og Objdok

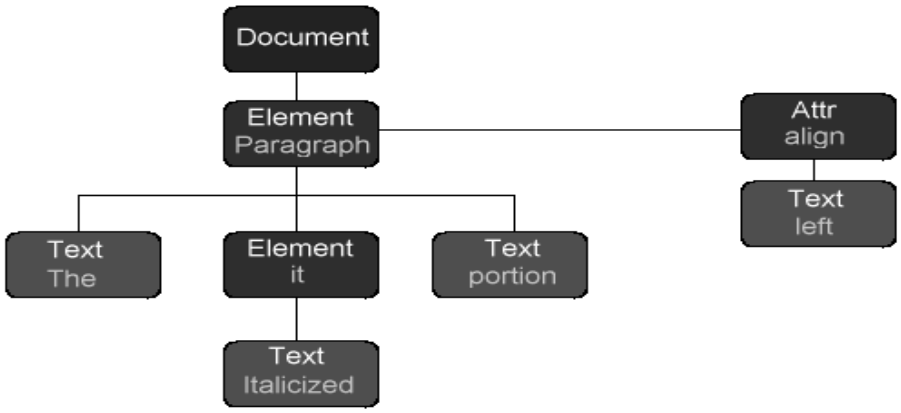
I arbeidet med **Obdok** forekommer **DOM**-modellen som sentral ved to anledninger. Den ene av disse er under **Obdok**-konverteringen, som omstrukturerer **Obdok**-instansedokumenter etter beskrivelsen i deklarasjonsfilen (se kapittel 6). Modellen opptrer også i presentasjonsdelen for visuell fremstilling av **Obdok**-dokumenter, da **XSLT**s datastruktur bygger på **DOM**.

3.4.1 DOM-eksempel

Eksemplet nedenfor og figur 3.1 belyser hvordan et **HTML**-dokument transformeres over til et **DOM**-tre. Eksemplet er hentet fra IBM XML zones sider[4]. #PCDATA og attributtverdier representeres i bladnoder, mens rotnoden tilsvarer **XML**s *dokument entitet*. Eventuelle tomme elementer ville også blitt representert som bladnoder.

```
<paragraph align="left">
  The
  <it>
    Italicized
  </it>
  portion.
</paragraph>
```

[4] http://www-106.ibm.com/developerworks/library/xml-perl2/



Figur 3.1: DOM eksemepelet som trestruktur

3.4.2 Sablotron

Sablotron (Sab n.d.) er et åpen kildekode (eng: open source) **XML**-verktøy fra Ginger Aliance som brukes til prosessering av **XSLT**, **DOM** og **XPath**.

Tilknyttet oppgaven benyttes **Sablotron** til transformeringen av **XML**-dokumenter til **HTML**-dokumenter. Dette som et ledd i visualiseringen av de konverterte **Obdok**-dokumentene. **XML**-dokumentet prosesseres mot **XSLT**-filens maler for transformasjonen, etter beskrivelsen i kapittel 3.3. Prosesseringen resulterer i en **HTML**-kode som kan tolkes direkte av en nettleser. Eksempler på visuell fremstilling av **Obdok**-kode finnes i kapittel 7.

3.5 Python

Under arbeidet med **Obdok**-konverteren (se kapittel 6 & appendiks A.3), ble programmeringsspråket Python (Pyt n.d.) benyttet. Det var tre hovedgrunner til å velge dette språket:

1. Python er et programmeringsspråk som er enkelt å lære. Den konkrete syntaksen er nært beslektet andre objektorienterte språk, og følelse av å mestre språket opparbeides raskt.
2. Språket er et skriptspråk og tilbyr således en *flying start* til programmeringen. Språket har også gode innebygde funksjoner som har vært til nytte i programmeringen av konverteren. Deriblant finnes regulære uttrykk og behandling av lister og tuppeler.

3. Python har dessuten et velutviklet *xml.dom*-bibliotek og er således et kraftig verktøy for arbeid med **XML**-strukturer.

Lett å Lære — Lett å Like Python er et skriptespråk som skal være lett å komme i gang med, både for nybegynnere og for de litt mer trenede programmerere. Som følge av dets deklarasjonsekskluderende beskaffenhet gir språket en *ad-hoc*-tilnærming på problemene, med de fordeler å ulemper dette innebærer. Pythonkode er dertil ofte kort og kompakt. Små oppgaver kodes i løpet av kort tid og språket er derfor svært effektiv til denne typen oppgaver. For større oppgaver, derimot, kan programstrukturen lett bli rotete og vanskelig å finne frem i. Python har en rekke biblioteks- eller innebygdefunksjoner det er likefremt å benytte. Deriblant finnes listefunksjoner, tuppel/hash-operasjoner og regulære uttrykk ("reg.exp").

Regulære Uttrykk Under arbeid og manipulasjon med tekstlige dokumenter, eksempelvis dokumentsentriske **XML**-dokumenter, vil behovet for søk, substitusjoner og splitting av tegnmønstre ofte melde seg. Regulære uttrykk er spesialdesignede verktøy for brukt til slike oppgaver. De kan gjenkjenne spesielle og generelle strenger med tegn — alfabetiske som spesialtegn — og tilbyr manipulasjon på disse. Syntaksen er i overbærende samsvar med programmeringsspråket C sitt *de facto* standard for tegnsett, i tillegg til språkets egne spesialtegn for grupperinger, listing, valgmuligheter og annen mønsteroppbygning.

***xml.dom.minidom*-biblioteket** Document Object Model (**DOM**) er, som beskrevet, en API fra W3C som tilbyr programmereren aksess og manipuleringsmuligheter på **XML**-dokumenter. Her representeres elementene i form av en trestruktur og muliggjør dermed navigasjoner og manipulasjoner som er umulige eller vanskelige å gjennomføre med en **SAX**-implementasjon eller utelukkende ved bruk av regulære uttrykk. Python har implementert en slik **DOM** i sitt *xml.dom*-bibliotek. Biblioteket er et velutstyrt og fullverdig verktøy både for lesing, manipulering så vel som bygging av **XML**-dokumenter. Dette biblioteket var således vitalt for utviklingen av **Obdok**-konverteren.

Kapittel 4

Byggesteiner & Strukturell Oppbygging av Obdok

Dette kapittelet tar for seg de grunnleggende byggesteiner og strukturer i **Obdok**, og sammensetning av slike for å skape objektorienterte dokumenter. Videre blir det forklart hvordan språket skiller mellom *deklarasjon* og *instansering* av dokumenter; herunder hvordan bestanddelene avgrenses i separate filer, og hvordan disse skal komponeres. Kapittel 5 vil så beskrive den konkrete **Obdok**-syntaks, mens kapittel 6 omhandler retningslinjer for bruk av de forskjellige verktøy som er produsert i forbindelse med fremstillingen av språket. Disse verktøyene konverterer, smelter sammen og presentere **Obdok**-dokumenter til én informasjonsbærende struktur.

4.1 Byggesteiner i Obdok

Om byggesteinene i et objektorientert programmeringsspråk er datatypene, vil de tilsvarende byggesteinene i **Obdok** være *informasjonstypene*. Det vil være mulig å dele inn informasjonstypene på en tilsvarende måte som datatypene blir inndelt i to klasser. Jamfør definisjon 2.3.1 og 2.3.2 på side 16, vil en slik klassifisering bestå av klassene *primitive* og *komplekse* informasjonstyper. Disse defineres i **Obdok** på følgende måte:

Definisjon 4.1.1. ***Primitive informasjonstyper*** *er de informasjonstypene som* ikke *er bygget opp av andre informasjonstyper (typisk elementer som kun består av #PCDATA eller billedfiler)*

Definisjon 4.1.2. ***Komplekse informasjonstyper*** *er de informasjonstypene som* er *bygget opp av en eller flere andre informasjonstyper (elementer som har en substruktur av andre elementer eller/også #PCDATA)*

Programmeringsspråkenes datatyper er lager for dataenheter. Primitive datatyper er begrenset til å ta vare på unære dataenheter, mens komplekse datatyper kan bestå av homogene eller heterogene sett av flere enheter av varierende sammensetning. Tilsvarende vil informasjonstypene "lagre", eller ta vare på, semantiske informasjonsenheter.

Definisjon 4.1.3. ***Informasjonstype*** *En informasjonstype er en representasjon av en semantisk informasjonsenhet.*

I **XML** vil en slik representasjon dreie seg om merking[1] eller innkapsling av informasjonsenheter (se forøvrig avsnitt 4.1.2 for representasjon av informasjonstyper). Primitive informasjonstyper vil i så måte være de informasjonstypene som kun kan kapsle inn én informasjonsenhet, mens komplekse informasjonstyper kan kapsle inn flere informasjonsenheter i ulik struktur.

Til informasjonstypebegrepet knytter det seg dog to problemer som må belyses. Det første er spørsmålet om hvorledes informasjonsenhet--begrepet skal tolkes; *hva* kan kalles en informasjonsenhet? Det andre problemet omhandler representasjonene av slike informasjonsenheter. *Hvordan* skal informasjonsenhetene representeres? Til slutt må informasjonstypene defineres, det vil si å navngi og bestemme egenskapene det sett **XML**-elementer som reflekterer dette kvantifisert informasjonsenhetsbegrepet.

4.1.1 Skop, Infon og `Set`

Artikkelen *Beyond Truth and Falsehood* (Hagen, Amdal & Nergaard 2000) tar for seg det tradisjonelle, faktabaserte informasjonsbegrepet og erstatter dette med et mer moderne, ikke-naivt epistemologisk *infon*-begrep; *«... we use the concept of the 'infon' as the unit of information, encoded in a transmission from sender to interpreter»*. Forfatterne kvantifiserer altså informasjon og presserer at slike enheter er *«... relative both to its scope and to its context»*.

En informasjonstype i kontekst, det vil si et **XML**-element plassert inn i **Obdok**-kode, vil fryse et slikt infon og gi det materielle egenskaper. Visa versa er det også først gjennom frysning av slike infon en kontekstramme vil bli skapt. Informasjonstypen `set`'s (se kap. 4.3.2) oppgave i **Obdok** er nettopp å opptre som en slik ramme. Konteksten, eller egentlig det subsett av denne en representasjon utgjør, avhenger altså av konstantenes kombinasjon. Konteksten kan dermed også sies å fryses mot materialisering.

[1] Mark-up

Skop og infon er tilsvarende knyttet til hverandre, ved at det er summen av (både de primitive *og* de komplekse) informasjonstypene som utgjør hvert skop eller synsvinkel.

4.1.2 Informasjonstyper og XML-Elementer

Til forskjell fra programmeringsspråkene, som jo kan oppbevare datatyper både som dynamiske variabler og statiske konstanter, *må* informasjonsenhetene i en **XML**-dialekt representeres i konstante lagringsenheter. Dette være seg om enhetene er kapitler, avsnitt og setninger, som i tradisjonelle dokumenter, eller informasjonstyper, som i **Obdok**-sammenheng. Disse konstante enhetene benevnes i **Obdok** som *konstanter*. Hver konstant tilsvarer én informasjonstype, og den beholder sin verdi hele sin levetid. **XML**-representasjonen av en primitiv informasjonstype vil i utgangspunktet være ett **XML**-element. Eksempelvis vil et formelt utsagn som en lov eller teori, kunne representeres med informasjonstypen `formal`. I eksemplene nedenfor vil elementet *<form_stat>* være en instans av denne informasjonstypen:

```
<form_stat>'loven'</form_stat>
```

I enkelte tilfeller vil det også være nødvendig å markere avsenderen av informasjonsenheten, komme med tekniske informasjon (som kan være beskjeder til **XSLT**-prosesseringens og lignende), eller merke elementene med et unikt navn/id. I slike tilfeller benyttes attributter. I loveksempelet finnes et attributt `form_name`. Dette benyttes til å gi navnet på den loven konstanten uttrykker. Tilsvarende vil en `picture`-konstant (her *bilde*) plassere url-referanser i et `src`-attributt.

```
<form_stat form_name="Ny lov #1">'loven'</form_stat>
          &
<bilde  src="url" />
```

Komplekse informasjonstyper representeres således ved å bygge opp forskjellige konstellasjoner av flere subelementer. Eksemplifisert vil en rekke av informasjonstypen `formal`, kunne settes sammen til en `array` av slike og danne en hel serie med lover, en lovsamling:

```
<form_array array_name="Nye Lover - 2003">
  <form_stat form_name="Ny lov #1">'loven'</form_stat>
  <form_stat form_name="Ny lov #2">'loven'</form_stat>
  <form_stat form_name="Ny lov #3">'loven'</form_stat>
</form_array>
```

Eksemplene over er forenklede og regelbetonte, men viser hvordan enkle, meningsbærende enheter kan kapsles inn i konstanter og representeres som **XML**-elementer. De viser til en én-til-én relasjon mellom *enheter* og *typer* som enkel å forholde seg til. I realiteten vil imidlertid ikke alle tilfeller være så enkle. Representasjonene av primitive informasjonstyper vil kunne inneholde et begrenset sett subelementer, dog uten at dette berører relasjonen (én-til-én) mellom informasjonstypen og informasjonsenheten/infon'et. Disse subelementene vil kun være der av kosmetiske eller funksjonelle årsaker.

Kosmetikk

Ovenfor er primitive informasjonstyper definert som *«informasjonstyper som ikke er bygget opp av andre informasjonstyper»*. Det er også forespeilet et samsvar mellom en primitiv informasjonstype og en **XML**-representasjon som *ikke* inneholder subelementer/substrukturer (slik de er spesifisert i dokumenttype-definisjonen). En slik fortolkning medfører likevel bare *delvis* riktighet, siden det må tillates **XML**-representasjonene av primitive informasjonstyper å kunne inneholde en begrenset mengde subelementer. Denne mengden vil bestå av *ikke-informasjonsbærende* elementer. Elementene vil være ** og ** (*stormskrift* og **uthevet**), *<note></note>*[2] samt *
* ('linjeskift') *<hr />* ('horisontallinje'). Elementnavnene, og fortolkning av disse, tilsvarer de beslektede kosmetiske elementene i **HTML**. Unntaket her er *'note'*-elementet som jo ikke finnes i **HTML**.

4.2 Funksjonalitet

Et **Obdok**-dokument er en strukturert sammenstilling av byggesteiner med et formål å formidle innhold. *Funksjonaliteten* til et slikt dokument vil dermed være kommunikasjon av dette innholdet — altså informasjon:

Definisjon 4.2.1. ***Funksjonalitet*** *defineres i* ***Obdok****-sammenheng til å være kommunikasjon av informasjon.*

Dynamiske Program — Statiske Dokument

Å programmere en datamaskin handler om manipulering av data — altså databehandling. Som beskrevet i kapittel 2.3.2 vil et programs funk-

[2] Fotnote

sjonalitet tilsvare summen av de operasjoner og kall som utføres på de datatyper og funksjoner programmet består av, og at dette er en hierarkisk modell. Et kjørende program er således i konstant dynamisk bevegelse og det er denne helhetlige bevegelse som gjør programmet.

Informasjonsbærende dokumenter er på sin side statiske konstanter, likeledes som at deres byggesteiner er statiske og konstante. Til tross for denne stillestående natur, åpner definisjon 4.2.1 opp for et funksjonsbegrep for **Obdok**-dokumenter. Dette funksjonsbegrepet sier at dokument har funksjonalitet når det fremstilles for en leser (eller annen mottaker). Denne kommunikasjonen opptrer som en dynamisk handling — i det leser mottar og tolker informasjon. Til forskjell fra dataprogrammer er det altså ikke manipulasjon av dokumentets byggesteiner som avgrenser dokumentets funksjonalitet, men manipulasjon av leseren[3]. Ved å ha innført et kvantifisert informasjonsbegrep, er det også mulig å si at den samlede funksjonalitet til et objektorientert dokument, er lik summen av funksjonaliteten til byggesteinene dokumentet er satt sammen av. En modulær inndeling av informasjonsdokumenter medfører således også en modulær inndeling av funksjonaliteten som inntar samme (hierarkiske) struktur som dokumentet. Dette tilsvarer programmers funksjonalitet beskrevet i kapittel 2.3.2 på side 17.

Grensesnitt

Nært beslektet til et dokuments funksjonalitet, ligger dokumentets grensesnitt. Et slikt grensesnitt forteller om forbindelsen mellom informasjonen, som én entitet, og leseren (eller en annen ANT-aktør) som den motsvarende entiteten. Denne forbindelsen defineres til å være:

Definisjon 4.2.2. ***Grensesnitt (informasjon)****: det språk eller den koding av informasjonen som kommuniseres.*

Som beskrevet i kapittel 4.1.1 forstås informasjonsbegrepet i **Obdok**-sammenheng som en materialisering av infon'et, introdusert i *Beyond Truth and Falsehood* (Hagen et al. 2000). I denne artikkelen argumenteres det også for en utradisjonell informasjonsprosess forskjellig fra den lineære *sender -> melding -> mottaker* modellen. Denne prosessen modelleres som en assosiert hendelsesstruktur, med mottaker som en aktivt deltagende aktør. For å kunne ha en informativ verdi blir et hvert infon kodet av informanten. Etter overføring dekodes så dette av mottaker. Både koding og dekoding påvirker infon'et, som dermed ikke er noe allmenngyldig og faktabasert, men en kontekstsensitiv enhet. Selve prosessen er også toveis, i det kodingen og respons påvirker begge parter.

[3] Merk: å manipulere leser ikke nødvendigvis å føre bak lyset, men å påvirke

En utfordring ligger dog i å definere et grensesnitt i et naturlig språk. For formelle språk er en slik oppgave nokså entydig og vil gi en eksplisitt løsning. For naturlige språk, er oppgaven derimot langt vanskeligere og mer implisitt. Joseph A. Gougen (Goguen 1994) foreslår å *"tørke"* underforstått (eng: tacit) eller kontekstsensitiv informasjon/kunnskap. En slik prosess vil tilnærme disse mot en mer formel natur. Charlotte Linde beskriver hvordan narrativ fremstilling kan binde sammen implisitt og eksplisitt informasjon (Linde 2001). Polyskopisk Modellering, på sin side, fremstiller informasjon i *forskjellige* grensesnitt. Grensesnittene er dermed avgjørende for å knytte de forskjellige skopene opp mot forskjellige kontekstuelle miljø.

4.3 Informasjonstypene

4.3.1 Primitive Informasjonstyper

Listen nedenfor beskriver de primitive informasjonstypene som er valgt ut som basisbyggesteiner i **Obdok**. Dette er informasjonstyper som, til tross for sin tradisjonelle tilhørighet, hører til i de fleste informasjonsbærende dokumenter. I tillegg vil dokumenttype-definisjonen til hver informasjonstype bli presentert, som et alternativ til den ordrike presentasjonen. Denne vil for alle typer inneholde attributtet `name`. Dette er ikke et brukerbestemt attributt, men et attributt som kreves for prosessering av dokumentet. **DTD**'en finnes forøvrig å lese i sin helhet i appendiks A.2 på side 97.

Merk også at for å skille informasjonstypenes navn fra tilsvarende uttrykk i de språk typene beskrives — deriblant norsk og engelsk, men også formelle språk — merkes disse informasjonstypene med en egen font: `navn`.

text Den enkleste, men også den mest generelle informasjonstypen som er definert, er `text`-typen. Konstanten representerer en ren, uformatert tradisjonell tekst og inneholder ingen brukerbestemte attributter.

```
<!ELEMENT text (#PCDATA | i | b | br | hr | link | note)* >
<!ATTLIST text name CDATA #REQUIRED>
```

ingress `ingress`-typen vil være en *abstrakt*, eller høynivå informasjonstype. Strukturelt vil konstanten være lik `text`-typen, men skilles ut som en egen informasjonstype på grunn av sin karakter som metainformativ. Navnet er kjent blant de fleste fra eksempelvis avisartiklene, mens

ordets opprinnelse stammer i fra latin for 'innledning' eller 'begynnelse' (Taule 1993).

```
<!ELEMENT ingress (#PCDATA | i | b | br | hr | link | note)* >
<!ATTLIST ingress name CDATA #REQUIRED>
```

head `head`-konstanten er den tekstlige informasjonstypen av høyest nivå. Konstanten er tenkt brukt til stikkord, ideonomer, overskrifter og lignende.

```
<!ELEMENT head (#PCDATA | i | b | br | hr | link | note)* >
<!ATTLIST head name CDATA #REQUIRED>
```

statement En `statement`-konstant er et uttrykk, et sitat eller et utsagn som kommer fra en avsender. I tillegg til å inneholde *hva* som blir sendt, altså selve uttrykket, må en slik erklæring også gi informasjon om *hvem* avsender er. Denne informasjonen legges i et eget `announcer`-attributt.

```
<!ELEMENT statement (#PCDATA | i | b | br | hr |
                     link |  note)* >
<!ATTLIST statement name CDATA #REQUIRED
          announcer CDATA #REQUIRED>
```

picture En `picture`-informasjonstype er ganske enkelt en referanse til et bilde. Dette være seg om det er en tegning, fotografi, et ideogram eller en annen form for billedgjøring. Siden det i første rekke handler om bilder som skal representeres og lagres i en computer, må referansen vise til en fil av et format som nettlesere kan forstå, for eksempel gif, jpeg eller png. `picture`-konstanten defineres som en tom tagg med fire attributter, hvor av ett, `src`-attributtet, er påkrevd. Alle fire attributtene tilsvarer sine navnebrødre i *img*-elementet i **HTML**. Av den grunn vil ikke attributtnavnene oversettes i konverteringen, men i stedet blir de skrivet likefremt ut.

- `src` — beskriver biledfilens plassering
- `width` — bildets vidde
- `height` — bildets høyde
- `alt` — gir muligheten til å gi bilde en kjapp beskrivelse eller en "sub-title"

Visualisering av informasjon er viktig for å nå frem med et budskap. I polyskopisk informasjon hjelper visualisering (spesifikt ideogrammer) til med å avgrense perspektivet

> «Information is not only a collection of verbal facts ... By using a visual technique ... we are able to define perspective» (Karabeg 2000*a*)

```
<!ELEMENT picture EMPTY>
<!ATTLIST picture src CDATA #REQUIRED
                  width CDATA #IMPLIED
                  height CDATA #IMPLIED
                  alt CDATA #IMPLIED
                  name CDATA #REQUIRED>
```

formal På lavere nivåer i det polyskopiske hierarki, trengs en mer formalistisk entitet. `formal`-konstanten, vil være en slik. Informasjonstypen representerer typisk et korollar, en lov, en definisjon eller et bevis. Også denne konstanten har et attributt `form_name` hvori typens navn, nummer eller lignende kan plasseres. Selve begrepsforklaringen innkapsles som elementets #PCDATA.

```
<!ELEMENT formal (#PCDATA | i | b | br | hr | link |
                  note)* >
<!ATTLIST formal name CDATA #IMPLIED
                 form_name CDATA #REQUIRED>
```

verse En poetisk fremstilling av informasjon vil også kunne være en primitiv type. Diktformen er allment kjent, også innenfor informasjonsformidling. Fra gammelt av ble mye lærdom og visdom formidlet og videreført til nye generasjoner gjennom diktformen. Eksempler på dette kan leses i Hovamål, eller i gresk gudedikting. `verse`-typen er en representasjon av en slik informasjonsenhet. Ønskes et dikt med flere vers, kan dette gjøres enten ved å definere flere separate `vers`-konstanter, eller disse kan struktureres som en sekvens av suksessive vers (se `array` under 4.3.2).

`verse`-konstanten er gitt et ikke-påkrevd attributt, `verse_name`, for indeksering eller navngiving av versene.

```
<!ELEMENT verse (#PCDATA | i | b | br | hr | link | note)* >
<!ATTLIST verse name CDATA #IMPLIED
                verse_name CDATA #REQUIRED>
```

4.3.2 Komplekse Informasjonstyper (Struktureringsprimitivene)

De komplekse informasjonstypene er definert som *de informasjonstypene som* er *bygget opp av en eller flere andre informasjonstyper*. Dette vil si at **XML**-representasjonen innholder informasjonsbærende substrukturrepresentasjoner av andre primitive eller komplekse informasjonstyper. Mer konkret følger beskrivelsen av de komplekse informasjonstypene — den formelle **DTD**-definisjonen, fra *obdok.dtd*, følger typebeskrivelsen umiddelbart.

array Den første komplekse informasjonstypen som beskrives er `array`-konstruksjonen. Dennes egenskaper er direkte nedarvet fra programmeringsspråkenes array. En `array`, eller en *rekke*, er en sekvens av andre primitive eller komplekse informasjonstyper. Et krav som settes til denne mengden, er at den må være homogen. En `array` kan således kun bestå av én informasjonstype, men selvsagt mange forskjellige instanser av denne. Der vil også være mulig å bygge opp `array`-konstruksjoner av andre `array`-konstanser. Dybden av slike, det vil si hvor mange nivåer av `array`-informasjonstyper som finnes inni hverandre, er ikke begrenset, men det laveste/innerste nivået *må* alltid bestå av en primitiv informasjonstype, en *set-referanse* eller en `link` 4.4.3.

Eksempler på bruk av `array`-typen kan være en rekke av tegninger, en tegneserie, eller en rekke av `verse`-konstanter, som tilsvarende vil representere et dikt. Et eksempel på en `array` bestående av nye `array`-representasjoner, kan være en tegneserie hvor hvert av bildene i sekvensen er kommentert med et vers. Om en `array` består av utsagn representerer denne en monolog, mens en `array` med to monologer kan gi en dialog. Likeledes vil to (eller flere) billedserier modellere parallellhistorier. Siden en `array` er sekvensiell, kan oppramslister også indekseres.

`array`-konstantene kan gis et valgfritt attributt `array_name` for navngiving o.l.

```
<!ELEMENT array ((array)* | (statement)* | (picture)* |
                 (link)* | (set)* | (ideonomy)* |
                 (verse)* | (column)* | (grapholog)* |
                 (poetic)* | (ingress)* | (text)* |
                 (head)*) >
<!ATTLIST array array_name CDATA #IMPLIED
                name CDATA #REQUIRED>
```

set Et `set` er en heterogen mengde uten en distinkt struktur. Et `set` kan i likhet til rekker både komponeres av primitive og komplekse informasjonstyper, men denne mengden kan, til forskjell fra `array`-informasjonstypens innhold, være hetrogen. `set`-typen er dermed den mest generelle informasjonstypen som er modellert i **Obdok**. Analogisk til objektorienterte programmeringsspråk, kan det sies at dette tilsvarer et objekt. En strengere tolkning av analogien vil nok hevde at det vil være mer korrekt å sammenligne `set`-konstruksjonen med (blant andre C's) *struct*-konsept. Dette fordi `set`-konstantene ikke inneholder funksjoner. Likevel — som en følge av definisjonen av funksjonalitet knyttet til objektorienterte dokument (def. 4.2.1), og siden `set`-informasjonstypene deler egenskaper med objekter (som for eksempel evne til arv) — føres sammenligningen mellom `set`-informasjonstypen og objekt? Et `set` fremstår således som en container for å lagre semantisk relaterte informasjonstyper — en kontekstuell ramme og en "fysisk" yttergrense for **XML**-elementene.

Til `set`-elementet er det angitt to attributter. Ett av disse, `set_name`, er påkrev og fungerer som prosessidentifikator og som overskrift i visualisering av dokumentet. Attributtet må således være unikt blant de ulike `set`-konstantene innenfor en fil eller struktur. Det andre attributtet, `ait`, er valgfritt og angir om settet er av en viss *abstrakt informasjonstype* (AIT). Fire forskjellige abstrakte informasjonstyper beskrives i kapittel 4.3.3. Ved å sette `ait`-attributtets verdi til en av disse abstrakte informasjonstypenes navn, defineres settet til å være av denne typen. Av dette følger også at dette `set` *må* ivareta en viss komposisjon og struktur, etter mønstre beskrevet nedenfor. Attributtet `ref` benyttes av *set-referanser* for å angi hvilke `set`-konstant referansen peker på, og beskrives mer utfoldende i avsnitt 4.4.3. De øvrige attributtene som er beskrevet i **DTD**'en er kun til for prosesseringen og skal ikke bestemmes av bruker.

```
<!ELEMENT set ANY>
<!ATTLIST set name CDATA #REQUIRED
              ID CDATA #REQUIRED
              ait CDATA #IMPLIED
              ref_from CDATA #IMPLIED
              ref CDATA #IMPLIED
              set_name CDATA #IMPLIED>
<!-- Merk at dette er samme definisjon
som for pekertypen set-referanse -->
```

Kapittel 7.1 på side 69 viser et enkelt eksempel hvor informasjonstypene som hittil er presentert er tatt i bruk.

4.3.3 Abstrakte Informasjonstyper — AIT

Abstrakte datatyper er beskrevet i kapittel 2.3.4 som generelle, implementasjonsuavhengige grensesnitt for spesifikke problem. Dette er problemer som opptrer i forskjellige former, men som forekommer så ofte at det altså er effektivt å modellere generelle løsninger for dem. Design av informasjonsbærende dokumenter byr ikke på samme type spesifikke og formaliserte problemer. Likefullt, modellering av fullstendige og konsistente perspektiv etter Polyskopisk Modellerings Metode kan også gjøres etter generelle løsninger. Disse løsningene er her gitt navnet *abstrakte informasjonstyper* og defineres som:

Definisjon 4.3.1. ***Abstrakt Informasjonstype****: En abstrakt informasjonstype er et grensesnitt av en spesifikk komposisjon.*

Ved å komponere generelle løsninger etter Polyskopisk Modellering, kan visse kvaliteter garanteres ved informasjonen som formidles. Dette er kvaliteter som går på de funksjonelle egenskapene ved dokumentet. En abstrakt informasjonstype er således et predefinert, polyskopisk grensesnitt. I **Obdok** er komposisjonen av fire slike definert, samtidig som språket naturligvis også tilbyr bruker en generell komposisjonsplattform til å skape egne grensesnitt.

For de fire abstrakte informasjonstypene er både komposisjonsform og struktur predefinert. Det vil si at komponisten tvinges til å velge et spesielt utvalg informasjonstyper når denne skal instansere en AIT. Hva som gjenstår fritt for denne, blir dermed navngiving og den generelle instanseringen (det vil si å fylle ut informasjonstypene med #PCDATA eller eventuelt andre informasjonstyper). At disse abstrakte informasjonstypene er predefinert i språket, resulterer i at de også er strukturelt begrenset, sammenlignet med en ADT's struktur. Samtidig understøtter AIT-konstruksjonene, i likhet med ADT'ene, utvidelser av modularitet. En AIT, slik den er modellert nedenfor, er i seg selv et polyskopisk subsett.

Fire generelle Løsninger

Påfølgende finnes beskrivelse og spesifikasjon av representasjonen til fire AIT'er. Det er to ting som er verdt å bemerke i denne sammenheng. For det første er ikke løsningene, eller representasjonene, som er vist nedenfor endelige løsninger for å representere slike AIT'er. Det finnes alltid flere forskjellige måter å representere disse i **Obdok**. Legg også merke til at denne listen på ingen måte er noen utømmende liste over mulige AIT'er. Dette følger av teorien som språket bygger på. Denne viser heller

ikke til noe endelig eller sluttet mengde fremstillinger eller synsvinkler til informasjonen. Derimot staker den bare en kurs for hvordan slike skal utvikles. En senere versjon av **Obdok** vil således kunne inneholde et bredere utvalg slike informasjonstyper, eller en mulighet for brukerdefinerte AIT'er. Eksempler på bruk av abstrakte informasjonstyper finnes i kapittel 7.2.2 på side 79.

column Denne strukturen innehar avisartikkelens egenskaper. `column`-AIT'ens komposisjon er tredelt: overskrift, ingress og brødtekst. Disse representeres i **Obdok** av de primitive informasjonstypene: `head`, `ingress` og `text`. (En mer sammensatt datatypeoppbygning av `column`-AIT'er kan eksempelvis bringe frem en objektorientert avis.)

```
<!ELEMENT column (head, ingress, text)>
<!ATTLIST column name CDATA #REQUIRED>
```

grapholog `grapholog`-informasjonsstrukturen består av et bilde eller et ideogram, samt en mengde overskrift- og tekstkonstanter assosiert til bildet. Disse er i **Obdok** representert en `head-array` og en `text-array`. Ideogram-representasjonen er alene rik nok til en fullstendig skildring av den virkeligheten eller abstraksjonen den beskrives, men ved å tilføye et valgfritt antall tekstlige konstanter, og gi dette en hierarkisk struktur, danner `grapholog` en polyskopisk informasjonstype i seg selv. Tekst og bilde fyller ut hverandre: *«Ideograms are pictures ... which represents ideas»*.

```
<!ELEMENT grapholog (picture, array, array)>
<!ATTLIST grapholog name CDATA #REQUIRED>
```

ideonomy Ideonomi er viten om ideer. En `ideonomy`-informasjonstype skal således formidle en slik *idé*. Informasjonstypens komposisjon består av en `formal`-konstant for presentasjon eller definisjon av ideen, samt en rekke av `text`-konstanter (`text-array`), hvis oppgave er å innkapsle en mer utbroderende, ordrik, laverenivås beskrivelse av ideonomet.

```
<!ELEMENT ideonomy (formal, array )>
<!ATTLIST ideonomy name CDATA #REQUIRED>
```

poetic Mens `grapholog`-strukturen har et visuelt høynivåledd med et assosiert lavnivå tekstledd, vil `poetic`-informasjonstypen være en lyrisk

høynivåfremstilling med en tilhørende, mer formell, begrunnelse. Strukturen bygges således opp av en `array` av `verse`-konstanter, samt en rekke `text`-konstanter (etter behov) for lavere nivås fremstilling.

```
<!ELEMENT poetic (array, array)>
<!ATTLIST poetic name CDATA #REQUIRED>
```

4.4 Relasjoner

Tidligere i denne oppgaven, i kapittel 2.3 på side 15, er det vist til én konkret og én abstrakt måte datatypene i objektorienterte programmeringsspråk kan relateres til hverandre. Den konkrete relasjonsmuligheten er basert på indirekte variabelaksessering via *pekere* eller *referanser*. Den abstrakte relasjonsmuligheten, på sin side, går gjennom *arv* av egenskaper og funksjonalitet. Et spørsmål som nødvendigvis vil dukke opp under design av et objektorientert dokumentspråk, er dermed: Hvilke relasjonsmuligheter kan opprettes for informasjonstypene i **Obdok**, og hvordan kan slike relasjoner implementeres og representeres i **XML**? Siden W3C ikke har spesifisert arv i sin **XML**-spesifikasjon må det eksplisitt defineres hvorledes slik arv skal fortolkes i **Obdok**-dokumenter. Som en analogi til pekere og referanser, har hypertekst og dokumentlinking lang fartstid på Internett og i **HTML**-dokumenter. Dessverre har dog ikke alle erfaringer med dette vært like gode. Å finne et alternativ til hypertekst er derfor også en viktig utfordring.

4.4.1 Arv i Obdok

Problemer med Arvingsbegrepet

En av de viktigste faktorene bak suksessen til objektorientert programmering, var innføringen av arvingsbegrepet. En subklasse kan arve egenskaper og funksjonalitet fra sin superklasse, samtidig som den kan spesialtilpasses ved skreddersøm av intern oppbygning og funksjonalitet. For **XML**-dokumenter er dog situasjonen annerledes. I spesifikasjonen fra World Wide Web Consortium finnes intet generelt arvingsbegrep eller noen beskrivelse for hvordan et slikt skal fortolkes. For å kunne bringe arv inn i objektorienterte dokumenter, må et slikt begrepsapparat derfor først utarbeides, samt at det må spesifiseres en entydig definisjon for arvelig oppførsel. Det første, viktige steget på denne veien, er å avklare spørsmålet om hvilke egenskaper hos et **XML**-element som skal kunne nedarves til andre elementer. Dette fordi alle informasjonstypene

i **Obdok** blir representert av **XML**-elementer. De potensielle arvingsentitetene som følger av den øvrige oppbygging, er dermed arv av:

- attributter
- attributtverdier
- substrukturer
- #PCDATA (elementenes tekstlig innhold)
- en kombinasjon av en eller flere av disse

I samme vending må det også bringes klarhet i spørsmålet om hvilke elementsnivå arv skal kunne forekomme mellom. Skal egenskapene utelukkende kunne nedarves langs stammen, fra et superelement til dets subelement(er), eller/også skal arv skje horisontalt i trestrukturen mellom søskenelementer.

Løsningen på disse spørsmålene følger nokså intuitivt som en følge av analogien mellom objektorientering innen programmeringsspråk, objektorientering innen dokumentspråk, og gjennom den representasjonsformen for informasjonstypene som er blitt beskrevet. Dette presenteres i det påfølgende.

Obdok Arver Arvingsegenskapene fra Objektorienterte Programmeringsspråk

Det er særlig to steg under utviklingen av **Obdok** som har muliggjort en tett analogi også når det gjelder arvingsbegrepet fra de objektorienterte programmeringsspråkene. Det første steget var å innføre informasjonstypebegrepet, å definere en bestemt mengde informasjonstyper — deriblant komplekse informasjonstyper — og å la hver av disse representeres av **XML**-elementer. En følge av dette var at relaterte, kvantifisert informasjonsenheter kunne bli samlet og avgrenset i containere, eller `set`. Et `set` har egenskaper i seg som gjør at det kan fungere som en mal for andre `set`, på en tilsvarende måte som et superklasse er en mal for et subklasse. I **XML**-representasjonen er det dermed kun elementenes (`set`-konstantenes) substrukturelle innhold som vil bli nedarvet. Den konkrete arvingsstien (eng: path) går dermed i det horisontale plan, sett gjennom **DOM**-modellen. Som tabell 5.2 på side 59 viser, befinner både lavere- og høyerenivå `set` seg på samme nivå i **DOM**-treet.

Det andre, viktige steget på vei mot arv, var å definere en egen syntaks for deklarasjon av **Obdok**-dokumenters innholdsfortegnelse, og å skille denne fra den konkrete, materialiserte **XML**-representasjonen. Denne

syntaksen, som er beskrevet i kapittel 5.1, er å se på som et skall utenpå **XML**-dokumentet. Ved å separere deklarasjon fra instansering, ble det mulig å la komponisten sette egne navn på sine instanser av informasjonskonstantene. Separasjonen muliggjør også en notasjon som tilbyr bruker også å kunne bestemme arvingsrekkefølgen mellom forskjellige `set`. Den begrensede begreps- og strukturmengden som deklarasjonssyntaksen utgjør, er både programmer- og manipulerbar. Dermed kan det også garanteres en korrekt forplantning av de arvelige egenskaper.

Per definisjon er det kun lavnivå`set` som kan arve andre `set`. I en deklarasjon av et lavnivå`set` kan det således refereres til det høynivås `set` som er arvingsmal for dette lavnivå `set`. Syntaksen for dette er beskrevet i tabell 5.1 og eksempler på dette finnes i kapittel 7.2.1 & 7.4.

4.4.2 Pekerfunksjonalitet

I avsnitt 2.3.1 vises det også til den konkrete måten datatyper i objektorienterte programmeringsspråk kan relateres til hverandre, nemlig vha. *peker*-relasjoner. Gjennom pekere, eller referanse, er det mulig å nå variabler, funksjoner og objekter på en indirekte måte. En følge av denne funksjonaliteten, er muligheten til å bygge opp forskjellige datastrukturer. Innen dokumentkomponering er kanskje *hypertekst*, eller *hypermedia*, den funksjonaliteten med mest nærliggende egenskaper til pekerens. Beklageligvis fyller ikke hypermedia pekerfunksjonaliteten fullt ut, men har isteden sider ved seg som kan assosieres med GoTo-satsens. Dette er egenskaper som må elimineres om informasjonen skal bli konsistent.

Hypertekst

Den rådende måten som finnes i dag for å knytte sammen elektronisk tekst, er gjennom *hypertekst*. Et annet begrep for dette er også linking av tekst eller dokumenter. Hypertekst tillater linking av tekst med forskjellige lagringslokasjoner slik at disse kan relateres til hverandre. Ved å utvide til *hypermedia* kan også grafikk, video og lyd innbefattes i linkingen (eller direkte i dokumentet). Det er i kraft av denne egenskapen at hypermedia *ligner* programmeringsspråkenes pekere. Verktøyet tilbyr brukere på begge sider indirekte aksess til annet steds beliggende enheter. Dette kan igjen benyttes til å bygge opp strukturer — tilsvarende de strukturer vi kjenner i fra algoritmeteori. Dermed bidrar hypermedia til å gjøre flate og endimensjonale dokumenter flerdimensjonelle og velstrukturerte, og det bygger opp under en modularisering av informa-

sjonen. World Wide Web Consortium[4], utrykker det som at hypertekst er *«text which is not constrained to be linear»*.

Opprinnelig skriver hypertekstbegrepet seg fra Ted Nelsons Xanadu-prosjekt (Xan n.d.), mens teknologien opphav stammer fra Vannevar Bush' Memex-maskin, presentert i artikkelen *«As We May Think»* (Bush 1945). I dag kan det trygt sies at **HTML** er *«lingua franca for publishing hypertext on the World Wide Web»* (HTM n.d.), og at hypermedia, som en grunnleggende kvalitet, er en viktig faktor til vevens raske vekst og omfang.

GoTo-Problemet

Det er dog *to* hovedproblemer som har dukket opp som følge av bruken av denne enveislinkingen som **HTML** tilbyr. For det første oppleves linkene mye på samme måte som hvordan GoTo-setninger oppleves i programkildekode (når disse leses). Utstrakt og ustrukturert bruk av linker kan lede til en *spagettistruktur* på dokumentmassen, og konsumering av disse fortoner seg ofte som rotete, uorganisert og komplekst. Fenomenet er gitt navnet *spagettilinking* (Lowe & Hall 1999); et begrep hvis opphav nettopp er hentet fra programmeringsverdenen. *Spagettikode* brukes om rotet og ustrukturert kode som er vanskelig å lese og forstå — kode gjerne er preget av GoTo-satser.

Den andre hovedbivirkningen med bruk av slike linker er at målet ofte endres, eller bortfaller fullstendig, etter linkens opprettelse. En slik *død* link vil da ende i en blindveie. En analogi til programmeringsverdenen her, vil være om målet eller adresselinjen til en GoTo-setning var blitt flyttet eller opphørt å eksistere. Da ville programmet antagelig endt i en "runtime error", eller i det minste resultert et uriktig program. På web-dokumenter blir ikke følgen av døde linker så fatale. En nettleser for web-dokumenter vil eksempelvis ganske enkelt bare gi tilbakemelding om at siden ikke lenger eksisterer. Informasjonen, derimot, vil på sin side få det samme fatale utfall som kjøring av program med døde GoTo-satser: den vil bli inkonsistent.

Åpne og Lukkede Miljø

Problemene beskrevet ovenfor er både nært tilknyttet hverandre, men også en direkte følge av hvordan Internetts sammensetting og struktur. Som en følge av sin nettverksarkitekturiske oppbygging, er Internett blitt en flat, demokratisk (semi-anarkistisk) organisasjon. Det finnes in-

[4] http://www.w3.org/WhatIs.html

gen enhetlig og unison overordnet makt eller kontrollorgan som kan sørge for at linker er konsistente og strukturen håndgripelig. Dette ansvaret hviler altså på den enkelte bruker, men som både Goguen, Karabeg og Lowe/Hall er inne på (Goguen 1997, Lowe & Hall 1999): nye løsninger er nødvendig for å stå bedre rustet på vei inn i informasjonsalderen.

I det følgende beskrives to informasjonssystem som har lykkes i å føre konsistens i både informasjon og i bruken av linking/pekere. Det ene av disse er hentet i fra en fiktiv SiFi-verden, mens det andre tilhører programmererens virkelighet. Dette for så å sammenligne med det informasjonssystem **Obdok** henvender seg mot — nemlig Internett.

Memory Holes — '1984' I sin fiksjonsroman '1984' (Orwel 1949), skriver George Orwell om arbeidere hvis oppgave var å erstatte innhold fra nyheter, bøker, filmer, manualer etc. Når innholdet i en artikkel, eksempelvis, ikke lenger skulle være til Partiets ønsker, måtte denne skrives om. Dette kunne være seg å dekke over tilintetgjørelsen av personer, om Partiet skulle skifte syn i en sak, eller om staten, Oceania, skulle veksle alliert partner. Hva enn endringen måtte være; arbeidernes oppgave var å skrive nye notiser, artikler, lage nye filmsnutter og lignende som passet bedre med den nye tilstanden. Det nye materialet ble sluppet ned i pneumatiske rør, som de kalte Memory Holes. Disse 'black-box'-maskinene erstattet gammelt innhold med det nye. Så lenge Partiet mestret å holde informasjonssystemet konsistent — at alle enheter som ble berørt av en endring også ble endret, og de enhetene som ble berørt av at de førstnevnte endringene ble endret osv. — kunne de også kontrollere alt av informasjon samfunnet skulle ha tilgang på. Så lenge de kunne holde informasjonen konsistent kunne de viske ut folks hukommelse og sådan også endre sannheten. Alt som var fantes av bilder, tekst, lyd og underbygde én mening, og var dermed gyldig som sann. Menneskers minne er flyktig, det trykte ord ble lov. *«At all times the Party is in possession of absolute truth, and clearly absolute truth can never have been different from what it is now».*

Systemet Orwell beskriver er et informasjonssystem som eliminerer oppdaterings- og blindveisproblematikken som oppleves på Internett. Informasjonssystemet er konsistent og til en hver tid fullstendig og "sant". *Hvordan* systemet overvåker alle berørte ledd av en endring, fortelles ikke, men det er naturlig å forestille seg at dette bygger på en slags Topic Map[5] — at et hvert emne, navn hendelse, sted, operasjon et cetera, er bundet, eller *linket*, sammen. Gjennomførbarheten av dette hviler på at systemet kontrolleres fullstendig av det politiske apparatet; et diktaturisk og absolutt apparat. Disse kontrollerer både informasjonens innhold

[5] Se eksempelvis `http://www.topicmaps.org/xtm/1.0/`

og flyt, og dermed også menneskene som arbeider med informasjonen. Hele aktør nettverket (Latour 1987, Law 1986) er et lukket, kontrollert og konsistent miljø.

Utviklermiljø Tilsvarende informasjonsbehandlingen i '1984', skjer utviklingen av programsystemer i enhetlige, lukkede miljø hvor utviklerne har absolutt, "diktatorisk" kontroll over prosessen. Beslutninger tas av ett organ, eventuelt en homogen organisasjon. Disse kan kontrollere alle entiteter systemet er bygget opp av, og at relasjonene mellom disse er korrekte. Dermed kan også et konsistent system garanteres. Rationals 'Unified Software Development Process' (Jacobsen, Booch & Rumbaugh 1999) er jo nettopp et eksempel på dette. Begrepet presserer det felles og enhetlige (eng: unified) som kjennetegner prosessen.

Internett — Et Åpent Miljø Internett kan beskrives som et superinformasjonssystem satt sammen av nettverk av subsystemer. Dette nettverket har en karakter som gjør kontroll vanskelig. Eksempelet med spesifiseringen og forsøk på å innføre IPv6 et godt eksempel til å belyse dette (Ole Hanseth 1998, kap. 2, 6). Nettopp som følge av sin nettverksstruktur (semi-anarkistisk) fører overgangen til en ny versjon av Internett til store problemer. Internetts strukturelle oppbygning medfører også, på et mer konkret plan, at kontroll av de enkelte nettverksentiteter blir umulig — så også kontroll av konsistens av linker og disses innhold. Hver enkelt distributør er fri til selv å produsere hva han vil, og til å koble dette opp mot hva han vil på det øvrige nettet. Problemet er at denne brukeren dog *ikke* er fri til å kontrollere at målet for de linkede siden fremdels er operative, levende og konsistente i forhold til sin side.

En naturlig følge av den beskrevne internettarkitektur, vil derfor være at det er umulig å garantere fullstendig konsistens for dette informasjonssystemet. Selv for en liten enhet, som en hjemmeside, vil jobben i å oppdatere og å kontrollere utgående linker fort bli stor. For større portaler eller nettaviser har oppgaven vist seg *for* stor. En fagretning som Webdesign forsøker å gjøre grep for å få bukt med problemene. **Obdok** vil ikke foreslå noen løsning på problemet med utgående linker, *men* språket presenterer en alternativ måte for relasjoner mellom interne entiteter.

4.4.3 Hybridlink

Selv om hypertekst og dokumentlinking har sine svake sider og bivirkninger, utgjør funksjonaliteten samtidig en viktig resurs for å øke infor-

masjonsmassen, tilgjengligheten til denne, og ikke minst informasjonens struktur. Det ville derfor være som å la en mulighet gå til spille, om ikke denne **Obdok**-spesifikasjonen skulle tilby bruker en måte å knytte dokumenter og resurser sammen. Samtidig er det viktig at løsningen som velges bygger opp under de polyskopisk verdier — presentert tidligere i denne oppgaven — og at den greier opp i problemene rundt spagettilinking og strukturering av tekst. Løsningen som presenteres er en hybridløsning bestående av *to* forskjellige link- eller pekertyper; én ekstern og én intern link.

Den eksterne linktypen er i **Obdok** gitt navnet `link` og deklareres og instanseres på samme måte som primitive informasjonstyper. Den interne pekerkonstanten, *set-referansen*, er derimot et tomt element hvis attributtverdi peker til et internt element; dette elementet vil være et av de øvrige `set` som befinner seg i dokumentet. Det opereres altså med en todelt løsning som skiller mellom pekere til andre dokumenter, og pekere til interne elementer.

Det er allikevel ikke resursenes plassering som er hovedårsaken til å velge en slik hybridløsning på pekerproblemet. Derimot åpner løsningen for en strukturoppbygging, en presentasjonsform og en nivådifferensiering som både er innovativ, som utnytter tidligere ervervede erfaringer fra objektorientert systemutvikling, og som er i tråd med prinsippene og kriteriene i fra Polyskopisk Modellering. Dette, som et ledd i å minske bruken av hypertekst/GoTo-løsninger, og dermed også redusere, eller i alle fall temme, problemene rundt spagettilinking og døde linker. Samtidig opprettholdes muligheten til å knytte andre kilder til sitt dokument.

Pekerpresentasjon

Under følger presentasjonene av elementkomposisjonen til de to pekertypene som er introdusert ovenfor; først i ordrik form, dernest som dokumenttype-definisjoner.

link Denne pekertypen er en omdøping av **HTML**s *'anchor'*-element (<a>-tagg). `link`-elementet er gitt et attributt, `src`, hvor verdien oppgir målet for linken som en url-adresse. Elementets *#PCDATA*-seksjon bestemmer hvor linken skal forankres; eksempelvis *Trykk Her!*

Det er to måter å instansere slike linker. Den ene går vanlig gjennom deklarasjon og instansering av konstantene. Dette vil føre til at konstanten opptrer som en selvstendig entitet i dokumentet. Videre kan `link`-konstanter implisitt deklareres i teksten med nøkkelordet *'link'*. Imp-

lisitte linker, vil forøvrig, føre til invalide[6] **XML**-struktur. Eksempelet i figur 7.4 på side 78 viser begge.

```
<!ELEMENT link ANY >
<!ATTLIST link src CDATA #REQUIRED
               name CDATA #REQUIRED>
```

set-referanser Pekere til andre `set`-informasjonstyper i samme dokument deklareres med nøkkelordet *'set'*, men omtales i **Obdok** som *set-referanser*[7]. Pekertypen er definert i dokumenttype-definisjonen til å være et tomt element. Det `set` pekeren refererer til angis i `ref`-attributtet. Verdien oppgis til å være `set_name`-verdien til mål-`set`'et. Høynivå `set` kan ikke være mål for referanser. Det er også en restriksjon at pekertypen må både deklareres og instanseres *før* det refererte `set`. En følge av dette er også at sykler er umulig. Derimot åpner denne pekertypen for strukturbygging på en utradisjonell måte, etter mønster fra både objektorientering og Polyskopisk Modellering. Spesielt til strukturer som lister og trær er *set-referanse* velegnet.

Som nevnt er *set-referanser* begrenset til *kun* å peke på `set`-informasjonstyper. Grunnen til dette er at pekertypen av funksjonelle årsaker må ha en strukturert informasjonstype som mål. Den trenger *både* et ankerpunkt og et kontekstuelt innhold festet til dette ankeret. Dette finnes (i de løsninger som er valgt for representasjon av informasjonstyper i **Obdok**) ikke i primitive informasjonstyper. Avgjørelsen om denne restriksjon forsterker også et objektorientert grensesnitt i språket.

Den formelle dokumenttype-definisjonen overlapper med definisjonen for informasjonstypen `set`, se kap. 4.3.2. I eksempel 7.4 på side 78 vises også et skjematisk eksempel over bruk av arv, linking og *set-referanser*. Tilhørende denne **Obdok**-koden er også en visuell fremstilling av eksemplet.

[6] Som i ikke-valid

[7] Begrepet *set-referanser* benyttes for å skille pekertypen `set` fra informasjonstypen med samme navn. Merk også at pekerbegrepet *set-referanse* fremstår i stormskrift, mens typebetegnelsen, `set` fremstår i vanlig informasjonstypefont.

Kapittel 5

Obdoksyntaks

5.1 Imperativt Dokumentspråk

Obdok er et *imperativt* dokumentspråk. Det vil si at komposisjonene dokumentet består av bindes imperativt til å følge visse brukerbestemte mønster. Denne beskaffenheten skiller språket fra tradisjonell dokumentkomposisjon. Hovedsaklig er det *to* forskjeller som er verdt å bemerke:

- **Obdok** krever deklarasjon av informasjonstyper
- deklarasjon og instansering av dokumentet holdes adskilt i to separate filer — således adskilles også semantikk og struktur

Deklarasjon av Obdok-Dokumenter

Definisjon 4.1.3 forklarer en informasjonstype som en representasjon av en semantisk informasjonsenhet. I **Obdok** materialiseres hver slik informasjonstype som en konstant, eller et **XML**-element. I tillegg til å være av en bestemt *type*, har hver konstant også et *navn* og et semantisk *innhold*. Begge de to sistnevnte verdiene gis av bruker. For at **Obdok**-konverteren (*obdok.py* appendiks 6) skal vite hvilke informasjonstype disse konstantene representerer, skilles deklarasjon og instansering fra hverandre. Under kjøring av konverteren identifiseres og substitueres konstantenes navn med informasjonstyper. Dette for at senere applikasjoner og stilsett skal kunne gjenkjenne konstantene, samt at den resulterende strukturen vil være et valid **XML**-dokument..

En annen viktig grunn til å kreve deklarasjon av **Obdok**-dokumenter, er fordi dette frigjør arbeidet med objektdokumentstrukturen, og skiller

det fra arbeidet med det semantiske innholdet. Denne egenskapen vil bli beskrevet nærmere nedenfor.

Instanser adskilt fra Deklarasjon

Den vanligste måte å avgrense deklarasjoner fra instanser innen programmering, er å legge disse til starten av en blokk, eller i alle fall før de reelle forekomster. I **Obdok**, derimot, må innholdsfortegnelsene spesifiseres i en separat deklarasjonsfil. Årsaken til dette er todelt. Først og fremst ivaretar en slik splittelse at instanseringsfilen kan opptre som et selvstendig og velformet **XML**-dokument. Å spesifisere en deklarasjon etter de prinsipper beskrevet tidligere, krever en egen syntaks. For ikke å blande denne syntaksen inn i instanseringsfilen, slik at sistnevnte mister sin velformethet, adskilles altså disse fra hverandre i to filer. En slik løsning er igjen med på å forenkle arbeidet med **Obdok**-dokumentene, da det finnes verktøy for både bygging og kontroll av validitet/velformethet til **XML**-dokumenter.

Den andre årsaken til å velge en slik separert løsning, er at dette også fremtvinger separert arbeidsfokus mellom konsentrasjon om de strukturelle aspekter, på den éne siden, og fokus på det kontekstuelle innhold, på den andre. En slik tankegang tilsvarer således ideen bak skillet mellom den visuelle fremstillingen (stilsettet), fra den logiske fremstillingen (den konkrete representasjon), som beskrevet i kapittel 3.1.2 på side 22. Det å splitte opp det logiske arbeidet i to fokus, kan med andre ord sees på som en utvidelse av arbeidsoppdelingen W3C selv innførte med **CSS** og **HTML 4.0**. Arbeid med **Obdok** foregår således i *tre* faser; arbeid med struktur, med semantisk innhold, og med visuell fremstilling.

5.2 Deklarasjonssyntaks med Kommentarer

Syntaksen for deklarasjon av **Obdok**-dokumenter er forsøkt lagt så tett som mulig opp mot deklarasjonssyntaksen i de mest brukte objektorienterte systemprogrammeringsspråkene. Dette for å hjelpe bruker til å tenke objektorientert.

Primært kan det sies at deklarasjonen av **Obdok**-dokumenter er todelt, hvor hver av de to avdelingene omfatter henholdsvis høynivå- og lavnivåinformasjon. Dette er en inndeling vi kjenner igjen i fra Polyskopisk Modellering (se kap. 2). For å markere ut de to avdelingene, merkes disse med nøkkelordenene *'high:'* og *'low:'*. De deklarasjoner som følger etter hvert av disse nøkkelordene, sies å befinne seg innenfor dette nivået. Høynivåsdeklarasjoner må komme før deklarasjoner av lavnivås

informasjonstyper, og forekomsten av nøkkelordet *'low:'* avslutter den førstnevnte avdeling. Hvert av de to nivåbolkene er igjen bygget opp av et antall `set`-deklarasjoner. Et `set` kan, som beskrevet i kapittel 4.3.2, inneholde alle de andre informasjonstypene, samt `link`-konstanter og referanser til andre `set`. Hver av disse deklareres med *<informasjonstype, navn>* konstellasjoner, etter mønsteret beskrevet i kapittel 4 eller appendiks A.2. Unntaket er `link`, som *kan* brukes implisitt i dokumentet. For hver av representasjonene, gjelder det at de er på samme informasjonsnivå som det *set* de er deklarert innenfor. I tabell 5.1 er deklarasjonssyntaksen definert på en mer formell måte. Tabellen gjør dette ved hjelp av et Extended Backus-Naur Form skjema (**EBNF**).

```
<deklarasjonsfil> ::= <high> <low>?
<high> :: = high: <high-sets>+
<low> ::= low: <low-sets>+
<high-sets> ::= set <name>{<set-dekl>*}
<low-sets> ::= set <arv>? <name>{<set-dekl>*}
<arv> ::= (name)
<set-dekl> ::= <prim-dekl>* | <set-ref>* | <array-dekl>* |
                <link>*
<prim-dekl> ::= <prim-inftype> <name>
<prim-inftype> ::= head | ingress | text | statement |
                    picture | verse | formal
<set-ref> ::= set <name>
<array-dekl> ::= array (<prim-dekl>* | <set-ref>* |
                    <array-dekl>* | <link>* <name>) <name>
<link> ::= link <name>
```

Tabell 5.1: Syntaks for deklarasjon av **Obdok**-dokumenter

Det er dog noen poeng som er verdt å legge merke til ved denne syntaksen:

- `array`-deklarasjon må være bygget opp av en homogen mengde konstanter. Altså, en `array` kan være bygget opp av hvilke som helst informasjonstype (inkludert `array` og *set-referanser*), så lenge **alle** enheter i denne rekken er av samme, homogene art.
- Definisjonen av *'name'* finnes beskrevet i W3C sin **XML**-spesifikasjon[1] (XML n.d.). Konstantnavnene, levert av bruker, må altså følge disse avgrensingene

[1] `http://www.w3.org/TR/2000/REC-xml-20001006\#NT-Name`

- Et konstantnavn kan ikke inneholde norske bokstaver o.l.[2] Det er derimot ikke til hinder for #PCDATA-avdelingene å inneholde slike tegn
- Det er kun laverenivå-`set` som kan arve høyerenivå-`set`. Dette kommer frem av syntakstabellen og er ellers spesifisert i kapittel 4.4.1. Parameternavnet i arvingsutsagnet henviser altså til det høynivå-`set` som er mal for denne arv `set`
- Kommentarer i deklarasjonsfilen rammes inn med /**kommentar*/

5.3 Instansering

Den konkrete materialisering, eller instanseringen, av disse informasjonstypene tas altså hånd om av **XML**-elementer. Det kreves således av instanseringsfilen at denne *både* samsvarer med **XML**-spesifikasjonen (XML n.d.), *og* med den assosierte deklarasjonen. En instansering som følger begge disse krav, vil dermed resultere i et valid **XML**-dokument. Mer om dette i kapittel 6.1.2.

Instanseringsfilen må altså være et *velformet* **XML**-dokument, men i tillegg kreves det at strukturen og elementnavnene følger de strukturene og konstantnavnene som ble deklarert i den tilhørende deklarasjonsfilen. Roten i slike dokument, det elementet som omfatter de øvrige element i dokumentet, er gitt navnet *'obdok'*. Inneholdt i dette elementet finner vi igjen nivåinndelingene fra deklarasjonen. Disse gjenspeiles representeres av *<high>*- og *<low>*-elementene. Hvert av disse inneholder igjen elementer som representerer `set`-informasjonstyper, som igjen inneholder elementer av de andre informasjonstypene. Tabell 5.2 viser den underliggende elementkomposisjon.

[2] **Sax**-parseren som pythonskriptet benytter seg av lar ikke et slikt tegnsett å passere.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<objdok>
  <high>
    <!– –high-level set– –>
  </high>
  <low>
    <!– –low-level set– –>
  </low>
</objdok>
```

Tabell 5.2: Fast struktur for instansering av **Obdok**-dokumenter

Kapittel 6

Obdok som Verktøy

I de foregående kapitelene er den generelle og den spesielle syntaksen for komponering og bruk av **Obdok**-dokumenter beskrevet. Det er også forklart hvorledes ulike strukturer kan bygges opp, blant annet gjennom bruk av de predefinerte AIT-konstruksjonene. Hva som derimot ikke er beskrevet, er hvorledes deklarasjon og instansering skal smelte sammen til ett dokument, samt hvordan dette dokumentet prosesseres for visuell fremvisning. Dette kapittelet tar for seg disse temaene.

6.1 Obdok Konvertering

I forrige kapittel ble det beskrevet hvordan og hvorfor deklarasjon og instansering holdes adskilt i forskjellige filer. Oppgaven har også brakt klarhet i dokumenttype-definisjonens rolle for **XML**-filer, og at dokumentspråket **Obdok** er tuftet på en **DTD**. Når dette kapittelet så skal beskrive prosessen som smelter deklarasjonsfilen, instanseringsdokumentet og dokumenttype-definisjonen sammen til ett strukturert **XML**-dokument, vinkles dette gjennom å beskrive det verktøy som benyttes. Sammensmeltingsprosessen er i **Obdok**-sammenheng gitt navnet *konvertering*, og verktøyet som utfører denne konvertering heter *obdok.py*. Dette er et pythonskript med *to* hovedfunksjonaliteter: innlesing & parsering, og substitusjon & strukturkonvertering. Resultatet mates resultatet til standard ut i form av en **XML**-struktur. Filanvisning til dokumenttype-definisjonen kan sendes med som opsjon; **DTD**'en plasseres da internt i strukturen før utmatingen.

Koden til dette pythonskriptet finnes å lese i appendiks A.3

6.1.1 Innlesing og Parsering

Det første steget av konverteringen er innlesing av parameterfilene. Disse parseres, og det opprettes en datastruktur for hver av kildene. Samtidig med denne parseringen foretas leksikal- og syntaksanalyse. Siden instanseringsfilen er et regulært **XML**-dokument, parseres denne med **DOM**-parseren i *xml.dom.minidom*-biblioteket. Resultatet av dette blir en trestrukturrepresentasjon av dokumentet, subsidiært en feilmelding ved syntaksfeil.

Deklarasjonsfilen, på sin side, tolkes av en spesialkonstruert parser, siden denne syntaksen er definert spesifikt for **Obdok**. Metodene *parse_set*, *create_set* og *parse_array* gjenkjenner, ved hjelp av regulære uttrykk, mønstre fra deklarasjonens syntaks (jfr. tabell 5.1 på side 57). Disse funksjonene genererer ulike objekter for de forskjellige informasjonstypene som parseres, og bygger opp en intern struktur ved hjelp av disse. Metodene blir kalt rekursivt, ettersom komplekse informasjonstyper kan nøstes; en rekke (`array`) kan bestå av nye rekker som (hver for seg) igjen kan bestå av nye rekker og så videre. Resultatet av denne parseringen blir én trestruktur for hvert `set` i dokumentet. Hvert slik tre inneholder en objektrepresentasjoner av de deklarerte informasjonstypene internt i hvert slikt `set`. Disse trærne danner grunnlaget for både elementnavnsubstitusjon, så vel som den strukturelle oppbygging i den endelige **XML**-strukturen.

Informasjonstypetre

Den informasjon deklarasjonen av et **Obdok**-dokument besitter, bevares i konverteringsskriptet igjennom opprettelsen av trær. Disse trærne ivaretar ikke bare data om konstantenes navn og informasjonstype, men også hvilket mønster disse vil ha i forhold til hverandre i instanseringen. For å få til dette bygges trærne opp over tre ulike objekttyper. Disse er listet opp nedenfor med navn og hvilke egenskaper de innehar. (Merk at alle klassene har en variabel som beskriver hva slags informasjonstype objektene representerer. Dette er lagret i variabelen 'ident').

- **set** inneholder en peker til det 'set'-objektet som representerer dennes super-`set` (som førstnevnte `set` arver fra). Denne pekeren kan være tom hvis dette `set` i seg selv er super-`set`. Klassen har også deklarert en tuppelliste hvor det kan finnes hvilke konstanter klassen inneholder ('prop{}').
- **array** inneholder hvilke informasjonstype rekken består av. Typeverdien finnes via 'prop{}'.

- **inftype** inneholder bare hvilke primitive informasjonstypen objektet representerer.

Property-tuppelen ('prop{}') er av en slik karakter at nøkkelen ivaretar konstantnavnet (*name*-verdien) til informasjonstypen, mens den relaterte tuppelverdien peker til informasjonstypeobjektet i deklarasjonstret.

Som nevnt opprettes et tre for hver `set`-forekomst i deklarasjonen. Disse trærne holdes rede på i en sekvensiell liste ('all_sets{}'). Denne listen har samme beskaffenhet som property-tuppelene med navn som nøkkel og peker til -*set*-objektet som relatert tuppelverdi. For alle trær i denne listen er det forøvrig verdt å merke seg at rotnoden alltid vil være et 'set'-objekt, mens bladnodene må være representasjoner av enten primitive informasjonstyper, *set-referanser* eller `link`-konstanter.

6.1.2 Translering til ny XML-Struktur

For at det konverterte dokumentet skal være et valid **XML**-dokument, må elementenes navn i instanseringen oversettes så de samsvarer med navnene i det elementsett som er definert i *obdok.dtd*, appendiks A.2. Denne substitueringen skjer i metoden *substitute*, som gjennomløper alle elementene i instanseringsfilen, finner igjen de tilsvarende deklarasjoner igjennom traverseringer av 'all_sets{}' og 'prop{}', og erstatter de aktuelle element- og attributtnavn. Dette skjer i metodene *substitute* og *sub_build*.

Frem til dette punkt har *set-referanser* kun henvist til det `set` denne referansen "peker" på. Løsningen for dette har vært at `ref`-attributtets verdi er identisk med `set_name`-attributtverdien i det `set` som blir referert til. For å aktivere denne pekeren, så å si, forer metoden *insert_ref* det refererte `set` inn som innhold i *set-referanse*-elementet. Til slutt sletter metoden *remove_set* den opprinnelige forekomster, for å ungå doble forekomster

Hvis det er oppgitt en dtd-parameteren under kjøring av konverterskriptet, vil dokumenttype-definisjonen settes inn internt i strukturen.

6.1.3 Spesialtegn

Legg merke til at **SAX**-parseren som benyttes i innlesing av **XML**-filen ikke gir støtte for spesialtegn som norske bokstaver. Disse oversettes således til usannsynlige strengkombinasjoner. Kombinasjonene er **HTML**s tegnreferanse entiteter[1], *uten* ampersandtegnet (&). Eksempelvis vil

[1] http://www.w3.org/TR/html4/sgml/entities.html

'Å' oversettes til *'Aring;'*, og *'ø'* til *'oslash;*. Av tekniske årsaker oversettes disse tilbake til opprinnelsen først i *obdok.php*-skriptet (se 6.2.4) ved dynamisk prosessering.

6.1.4 Kjøring av *obdok.py*

På UiO's maskiner kjøres *obdok.py* med *python2* på Linux plattformene. Skriptet kalles med tre argumenter, hvorav det siste er valgfritt:

```
-- -->python2 obdok.py <deklarasjon.dk> <instansering.xml>
      <dtd>?
```

Utmatingen fra dette skriptet vil mates til standard ut, men det vil ofte være hensiktsmessig å omdirigere (eng:redirect) dette til fil. I det videre vil en slik fil også bli kalt for *utmatingen*.

6.2 Visualisering

Det er tidligere i denne også oppgaven beskrevet hvordan **XML**-dokumenter skiller seg fra forgjengeren **HTML** i å holde logisk struktur adskilt fra fysisk presentasjon. Stilsettspråket **CSS** er nevnt som et alternativ til å spesifisere den visuelle presentasjonen, transformasjon gjennom **XSLT** en annen mulighet. Det er også mulig å sette sammen flere, forskjellige stilsett til en struktur, siden disse jo er skilt fra hverandre. En hver bruker av **Obdok** kan med andre ord bestemme sitt eget stilsett til de dokumentene de måtte komponere, i tillegg til den **XSLT**-transformasjonen som er laget i forbindelse med denne oppgaven.

6.2.1 Obdok–Stilsettet

Tilknyttet denne versjonen av **Obdok** er det altså implementert et stilsett. Dette er i hovedtrekk en **XSLT**-transformasjon til **HTML** som også til dels benytter seg av **CSS**-standarden. Filene heter *obdok.xslt & cstyle.css*, og koden for dette er lagt ved i appendiks A.4.1 & A.4.2.

Transformasjonen gjennomløper alle forekomster av `set`-elementer i utmatingen sekvensielt. For hvert `set` blir alle de inneholdte informasjonstypene prosessert — også dette vil skje etter rekkefølge. Det dykkes rekursivt ned i hver konstant for å finne eventuelle kosmetiske elementer ('kosmetiske elementer' er beskrevet i kapittel 4.1.2). Også `array`-konstanter og *set-referanser* blir gjennomløpt rekursivt for å få med alle de inneholdte informasjonstyper.

Hvis et `set` er av en abstrakt informasjonstype (sjekkes på `ait`-attributtverdien), får denne en særbehandling i prosesseringen. Avhengig av hvilke AIT dette er, prosesseres dette `set` av bestemte templates. Disse kalles ved uttrykket *call-template*. I de bestemte templatene prosesseres så disse etter det grensesnitt disse representerer.

6.2.2 OnClick

I forbindelse med pekerfunksjonaliteten *set-referanse*, er det lagt et javaskript til visualiseringen. Dette heter *onclick.js* og koden for dette kan leses i appendiks A.5. Skriptet er plassert i hodet (eng: header) til **HTML** presentasjonen av **Obdok**-filene. Funksjonaliteten til javaskriptet er å skjule blokker, for å vise disse (igjen) på kommando fra leser. Effekten av dette er en innbilt import av innhold til dokumentet. Når bruker trykker med venstre musknapp på ankerpunktet til den skjulte blokken, kommer resten av blokkens innhold til syne innen for ankerpunktets ramme. Bruker blir altså innbilt at en links innhold flettes inn i dokumentet, når dette i realitet har vært tilgjengelig hele tiden; bare skjult av javaskriptet.

I kapittel 4.4.3 argumenteres det for innføringen av en intern linktype av struktureringsmessige årsaker; både for utviklings- og for presentasjonsprosessen. Ved hjelp av *onclick.js*-skriptet, ikke bare oppfyller **Obdok**-visualiseringen de argumenter som i kapittel 4.4.3 ble presentert, men fremstillingen gjøres på en måte som samtidig er gjenkjennelig for brukere familiære med tradisjonell hypertekst. Det er dog to begrensninger i bruken av disse linkene. For det første begrenses *set-referanser* til bare å peke på `set`-informasjonstyper. Dernest, for at dette skal virke optimalt, må siden leses fra en nettleser som støtter **DOM**-level 2 og **CSS2**-level 2. Disse benyttes i *onclick.js*-skriptet til å skjule/vise blokknoder (`set`-informasjonstyper representert i **HTML** som *'div'*-elementer).

En senere utvidelse av **Obdok** vil også kunne hente informasjon fra fjerne lokasjoner gjennom utgående linker (`link`-informasjonstype) for å plassere og skjule dette på tilsvarende måte som det her er gjort med *set-referanser*. Mer om dette i kapittel 8.2.4.

6.2.3 Sablotron-Prosessering

Prosesseringen av **XML**- og **XSLT**-dokumenter til **HTML**-kode, foretas av en Sablotron-parser. Denne parseren er beskrevet i kapittel 3.4.2. Sablotron-parseren kjøres på UiOs systemer med kommandoen[2]

[2] Se forøvrig sabchmd – –help

```
-- -->sabchmd <xslt> <xml> <html>?
```

på Solaris plattform. Utmatingen skjer til standard ut, men kan omdirigeres til som en **HTML** fil ved å angi filnavn som tredjeparameter i kallet.

6.2.4 Prosessert Fremvisning

Tilknyttet Sablotron-prosesseringen er det også opprettet et PHP-skript som sørger for dynamisk **HTML**-utmating til nettleser. Dette skriptet kalles med to parametere i *dekl* og *inst*, som viser til henholdsvis **Obdok**-deklarasjonsfilen og -instanseringsfilen. Skriptet kaller *obdok.py*-konverteren, forer dennes utmatningen inn i en Sablotron-prosessering opp mot *obdok.xslt*-stilsettet, og resulterer altså i dynamisk **HTML**-kode som nettleser kan forstå. Koden kan leses i appendiks A.4.3 på side 110.

Eksemplene som er lagt ved kapittel 7 kan ses i sin helhet på siden `http://commisarius.ifi.uio.no/~hansogj/Hovedfag/Oppg/main.php`. Siden videreformidler leser videre til de konkrete eksempler.

Kapittel 7

Eksempler

Dette kapitlet tar for seg fire eksempler med bruk **Obdok** til komposisjon av informasjonsbærende dokumenter. I tillegg til å være nivåregulert på kompleksitet, er også eksemplene delt inn i to klasser som skiller mellom dokumentenes funksjonalitet. To av eksemplene er således skjematiske og simplistiske, mens de to øvrige representerer bruk av **Obdok** til modellering av informasjon. Tilknyttet hvert eksempel er et bilde som viser hvordan disse kan se ut etter at filene er prosessert, tillagt stilsett, og tolket av en nettleser. Disse bildene er utklipp fra en side som er laget i forbindelse med utarbeidelsen av dokumentspråket. Denne siden kan i sin helhet sees på URL-adressen: `http://commisarius.ifi.uio.no/~hansogj/Hovedfag/Oppg/main.php`. Siden anbefales å lese fra en Mozilla 5.0 nettleser (eller nyere), som følge av den aktive bruk av **DOM**-level 2, og **CSS2**-level 2 – avhengige funksjoner.

7.1 Høynivåeksempler

7.1.1 Skjematisk Eksempel av Informasjonstypene i Obdok

Det første eksempelet som vises er et enkelt tilfelle for å vise de ulike informasjonstypene satt i bruk. Eksemplet inneholder én forekomst av hver type, i tillegg til at rekken *'tekster'* er bygget opp av multiple forekomster av `text`-konstanter. Eksempelet viser også kommentarer i deklarasjonssyntaksen.

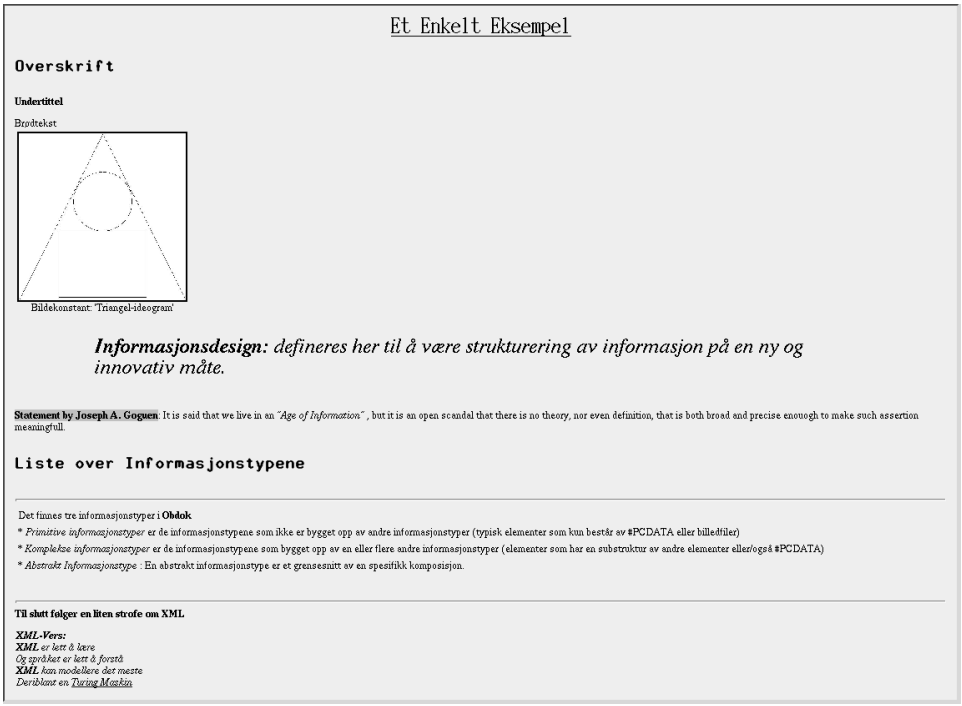
I tillegg til å vise de ni informasjonstypene, er eksemplet også en lettfattelig innføring i anvendt **Obdok**-kode. Det er lett å følge deklarasjon og materialisering av denne gjennom instansering i **XML**-filen. Figur 7.1 billedgjør en visualisering av eksemplet.

forste_eks.dk

```
/*Dette er kun en enkelt eksemplifisering av hvorledes et \\
objektdokument kan deklareres */

high:
set et_set{
    head overskrift;
    ingress undertittel;
    text broedtekst;
    picture triangel;
    formal infdes;
    statement goguen;
    array (text tekst_stubb) tekster;
    verse xml_vers;
}
```

forste_eks.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!-- Dette er kun en enkelt eksemplifisering av hvorledes
et objektdokument kan instanseres -->
<obdok>
  <high>
    <et_set set_name="Eksempel med bruk av Informasjonstyper">
      <overskrift>Overskrift</overskrift>
      <undertittel>Undertittel</undertittel>
      <broedtekst>Brødtekst</broedtekst>
      <triangel
        src="http://folk.uio.no/hansogj/Hovedfag/Oppg/Present/triangel.jpg"
        width="200" height="200" alt="Bildekonstant: 'Triangel-ideogram'" />
      <infdes form_name="Informasjonsdesign">
        defineres her til å være strukturering av informasjon på en ny og innovativ måte.
      </infdes>
      <goguen announcer="Statement by Joseph A. Goguen">
        It is said that we live in an <i>"Age of Information"</i>, but
        it is an open scandal that there is no theory, nor even
        definition, that is both broad and precise enouogh to make
        such assertion meaningfull.
      </goguen>
      <tekster array_name="Liste over Informasjonstypene">
        <tekst_stubb>Det finnes tre informasjonstyper i <b>Obdok</b></tekst_stubb>
        <tekst_stubb>*<i>Primitive informasjonstyper</i> er de
          informasjonstypene som ikke er bygget opp av andre
          informasjonstyper (typisk elementer som kun består av
          #PCDATA eller billedfiler)</tekst_stubb>
        <tekst_stubb>*<i>Komplekse informasjonstyper</i> er de
          informasjonstypene som bygget opp av en eller flere andre
          informasjonstyper (elementer som har en substruktur av andre
          elementer eller/også #PCDATA)</tekst_stubb>
        <tekst_stubb>*<i>Abstrakt Informasjonstype</i>: En abstrakt
          informasjonstype er et grensesnitt av en spesifikk
          komposisjon.</tekst_stubb>
      </tekster>
      <undertittel>
        <hr />
        Til slutt følger en liten strofe om <b>XML</b>
      </undertittel>
      <xml_vers verse_name="XML-Vers"><br />
        <b>XML</b> er lett å lære<br /> Og språket er lett å forstå
        <br /> <b>XML</b> kan modellere det meste<br /> Deriblant en
        <link src="http://www.unidex.com/turing/tmml.htm"> Turing
        Maskin</link>
      </xml_vers>
    </et_set>
  </high>
</obdok>
```



Figur 7.1: Enkelt eksempel for å vise primitive og komplekse informasjonstyper.

7.1.2 Ben & Arthur – Dialogen

Eksempel nummer to tar for seg et stykke tekst fra Dino Karabegs utkast til boken *What's going on?*. Karabeg har modellert informasjon gjennom en dialog, for på denne måten argumentere for behovet for kulturelle endringer. Eksempelet viser hvordan **Obdok** åpner for forskjellige representasjoner av dialogen. Det illustrerer dessuten hvordan informasjonstypene sammen kan skape en kontekstuell ramme for informasjon, og hvordan dokumentets struktur forenkles gjennom objektorientert komposisjon. Merk at begge modelleringene befinner seg i samme fil. Dette er naturligvis gjort utelukkende for å belyse av de nevnte egenskaper. Figurene 7.2 og 7.3 viser eksemplene etter prosessering.

arthur_ben.dk

```
/*Deklarasjonsfil for arthur_ben-konversasjon*/

high:
set ben_arthur{
    set parref;
    set linref;
    text author;
    head tittel;
    }

low:
set linear{
    array(statement stat) lin;
    }

set parallell{
    array(array(statement stat) stat1, array(statement stat) stat2) par;
    }
```

arthur_ben.xml

```
<?xml version="1.0" ?>
<obdok>
  <high>
    <ben_arthur set_name="Introductory dialog">
      <tittel>Subject and Method</tittel>
      <linref ref="Linear" />
      <parref ref="Parallel" />
      <author>
        This dialog is written by Dino Karabeg,
        and is assembled from the book draft <i>What's going on</i>
      </author>
    </ben_arthur>
  </high>
  <low>
    <linear set_name="Linear">
      <lin>
        <stat announcer="Ben">I don't understand this at all! Didn't
        you say that this was a serious book about culture?  That it
        <i> really </i> says what's going on?</stat>

        <stat announcer="Arthur">Yes, I did. </stat>

        <stat announcer="Ben">What is this story doing here then?</stat>

        <stat announcer="Arthur"> That story is a
        parable. Mrs. Adelaide's house is a metaphor for modern
        culture. </stat>

        <stat announcer="Ben"> So the author tells us what is <i>
        really </i> going on by telling us a story?  </stat>

        <stat announcer="Arthur"> The main message of the book is that
        we are, in effect, living in a house with failing
```

```
      foundations. That 'house' is our culture.  </stat>

      <stat announcer="Ben"> What do you mean by that? How can a
      culture be like a house with failing foundations? </stat>

      <stat announcer="Arthur"> Well, to see how you really need to
      read the book. The whole book is a <i> justification </i> or a
      proof of that message. The analogy between the house with
      failing foundations and the culture is shown point by point.
      </stat>

      <stat announcer="Ben"> What does that mean, that main
      message?</stat>

    <stat announcer="Arthur"> A house with failing foundations is
    not a place where one can live. It is unsafe. Doing repaires is
    also meaningless. It is necessary to rebuild the foundations.
    </stat>

    <stat announcer="Ben"> All right, all right, I know about the
    house, but what is the message about the <i> culture</i> ?
    </stat>

    <stat announcer="Arthur "> It's the same: That our culture is
    not a safe place to live in. And that trying to fix problems or
    to do small improvements is no better than patching holes in a
    house with failing foundations.  </stat>

    <stat announcer="Ben"> That does sound pretty serious. I
    thought that story was a joke.  </stat>


    <stat announcer="Arthur"> It's not a joke. I am telling you
    that book is serious. You should not be deceived by its
    light-hearted style. I think that the author wanted to put
    sugar coating around his message, to make it easier to
    swallow. But the message itself is completely serious. </stat>

    <stat announcer="Ben"> A bit <i> too</i> serious, for my
    taste. Is this perhaps another doomsday prophesy?</stat>

    <stat announcer="Arthur"> No, not at all. The message of the
    book is, in fact, optimistic. I would even say <i> unusually
    </i> optimistic, considering that the book talks about the
    prospects of our culture.  </stat>

    <stat announcer="Ben"> Optimistic? Are you calling the idea of
    living in a house with failing foundation <i> optimistic</i> ?
    </stat>

    <stat announcer="Arthur"> Yes, I am. To see why, think again
    about the metaphor: "The foundations are failing" is an
    identifiable situation which has a solution -
    reconstruction. We have all but given up trying to make a
    technological culture that can last. The book says it's
    possible. And a lot more than that: When a house is
    reconstructed, the result is usually not only safer, but also
    <i> better</i> ... </stat>

    <stat announcer="Ben"> Wait a minute, Arthur. You are taking it
    for granted that the modern culture really <i> is</i> like a
    house with failing foundations, which I don't. I don't believe
    the story. </stat>

    <stat announcer="Arthur"> You don't?  Well, that is just
    excellent. Then you are exactly the kind of person this book
    has been written form.  </stat>

    <stat announcer="Ben"> What do you mean? </stat>

  </lin>
</linear>

<parallell set_name="Parallel">
  <par array_name="DoubleStory">

    <stat1 array_name="Bens Story">

    <stat announcer="1 Ben"> I don't understand this at all! Didn't
    you say that this was a serious book about culture?  That it
    <i> really </i> says what's going on?  </stat>

    <stat announcer="2 Ben"> What is this story doing here then?
    </stat>
```

```
  <stat announcer="3 Ben"> So the author tells us what is <i>
  really </i> going on by telling us a story?  </stat>

  <stat announcer="4 Ben"> What do you mean by that? How can a
  culture be like a house with failing foundations?</stat>

  <stat announcer="5 Ben"> What does that mean, that main
  message?</stat>

  <stat announcer="6 Ben"> All right, all right, I know about the
  house, but what is the message about the <i>culture</i> ?
  </stat>

  <stat announcer="7 Ben"> That does sound pretty serious. I
  thought that story was a joke.  </stat>

  <stat announcer="8 Ben"> A bit <i> too</i> serious, for my
  taste. Is this perhaps another doomsday prophesy?</stat>

  <stat announcer="9 Ben"> Optimistic? Are you calling the idea
  of living in a house with failing foundation <i>
  optimistic</i>?</stat>

  <stat announcer="10 Ben"> Wait a minute, Arthur. You are taking
  it for granted that the modern culture really <i> is</i> like a
  house with failing foundations, which I don't. I don't believe
  the story. </stat>

  <stat announcer="11 Ben"> What do you mean? </stat>

</stat1>
<stat2 array_name="Arthurs Story">

  <stat announcer="1 Arthur"> Yes, I did. </stat>

  <stat announcer="2 Arthur"> That story is a
  parable. Mrs. Adelaide's house is a metaphor for modern
  culture. </stat>

  <stat announcer="3 Arthur"> The main message of the book is
  that we are, in effect, living in a house with failing
  foundations. That 'house' is our culture.  </stat>

  <stat announcer="4 Arthur"> Well, to see how you really need to
  read the book. The whole book is a <i> justification </i> or a
  proof of that message. The analogy between the house with
  failing foundations and the culture is shown point by point.
  </stat>

  <stat announcer="5 Arthur"> A house with failing foundations is
  not a place where one can live. It is unsafe. Doing repaires is
  also meaningless. It is necessary to rebuild the foundations.
  </stat>

  <stat announcer="6 Arthur "> It's the same: That our culture is
  not a safe place to live in. And that trying to fix problems or
  to do small improvements is no better than patching holes in a
  house with failing foundations.  </stat>

  <stat announcer="7 Arthur"> It's not a joke. I am telling you
  that book is serious. You should not be deceived by its
  light-hearted style. I think that the author wanted to put
  sugar coating around his message, to make it easier to
  swallow. But the message itself is completely serious. </stat>

  <stat announcer="8 Arthur"> No, not at all. The message of the
  book is, in fact, optimistic. I would even say <i> unusually
  </i> optimistic, considering that the book talks about the
  prospects of our culture.  </stat>

  <stat announcer="9 Arthur"> Yes, I am. To see why, think again
  about the metaphor: "The foundations are failing" is an
  identifiable situation which has a solution -
  reconstruction. We have all but given up trying to make a
  technological culture that can last. The book says it's
  possible. And a lot more than that: When a house is
  reconstructed, the result is usually not only safer, but also
  <i> better</i> ... </stat>

  <stat announcer="10 Arthur"> You don't?  Well, that is just
  excellent. Then you are exactly the kind of person this book
  has been written form.  </stat>
</stat2>
```

```
   </par>
  </parallell>
</low>
</obdok>
```

Ben: I don't understand this at all! Didn't you say that this was a serious book about culture? That it *really* says what's going on?

Arthur: Yes, I did.

Ben: What is this story doing here then?

Arthur: That story is a parable. Mrs. Adelaide's house is a metaphor for modern culture.

Ben: So the author tells us what is *really* going on by telling us a story?

Arthur: The main message of the book is that we are, in effect, living in a house with failing foundations. That 'house' is our culture.

Ben: What do you mean by that? How can a culture be like a house with failing foundations?

Arthur: Well, to see how you really need to read the book. The whole book is a *justification* or a proof of that message. The analogy between the house with failing foundations and the culture is shown point by point.

Ben: What does that mean, that main message?

Arthur: A house with failing foundations is not a place where one can live. It is unsafe. Doing repaires is also meaningless. It is necessary to rebuild the foundations.

Ben: All right, all right, I know about the house, but what is the message about the *culture* ?

Arthur: It's the same: That our culture is not a safe place to live in. And that trying to fix problems or to do small improvements is no better than patching holes in a house with failing foundations.

Ben: That does sound pretty serious. I thought that story was a joke.

Arthur: It's not a joke. I am telling you that book is serious. You should not be deceived by its light-hearted style. I think that the author wanted to put sugar coating around his message, to make it easier to swallow. But the message itself is completely serious.

Ben: A bit *too* serious, for my taste. Is this perhaps another doomsday prophesy?

Arthur: No, not at all. The message of the book is, in fact, optimistic. I would even say *unusually* optimistic, considering that the book talks about the prospects of our culture.

Ben: Optimistic? Are you calling the idea of living in a house with failing foundation *optimistic* ?

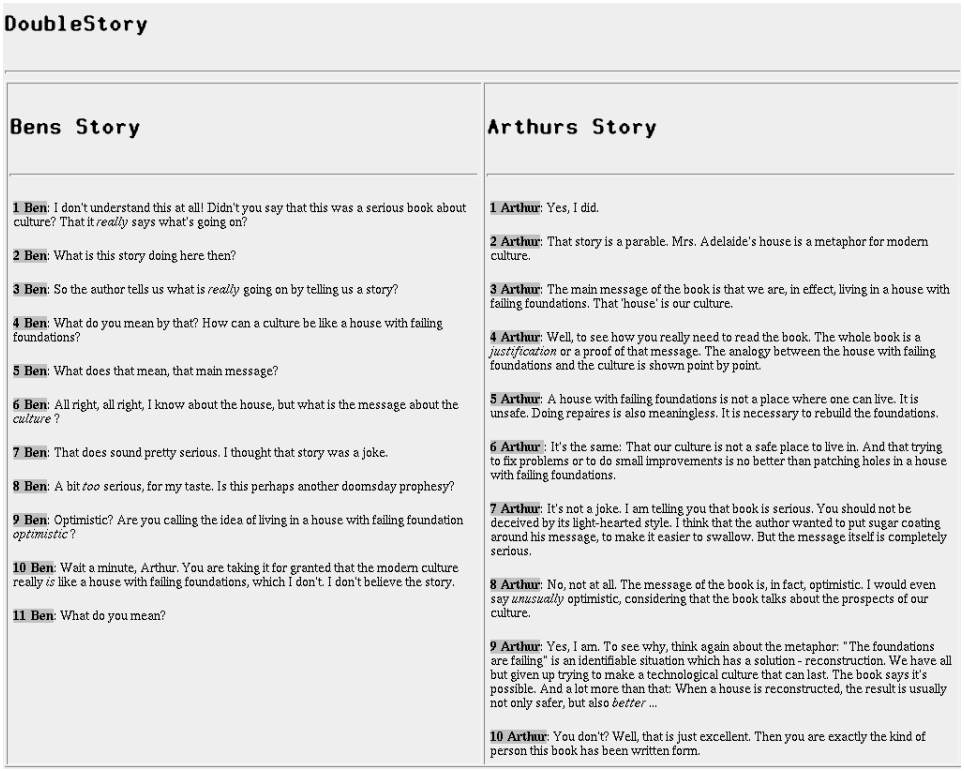
Arthur: Yes, I am. To see why, think again about the metaphor: "The foundations are failing" is an identifiable situation which has a solution - reconstruction. We have all but given up trying to make a technological culture that can last. The book says it's possible. And a lot more than that: When a house is reconstructed, the result is usually not only safer, but also *better* ...

Ben: Wait a minute, Arthur. You are taking it for granted that the modern culture really *is* like a house with failing foundations, which I don't. I don't believe the story.

Arthur: You don't? Well, that is just excellent. Then you are exactly the kind of person this book has been written form.

Ben: What do you mean?

Figur 7.2: Ben & Arthurs konversasjon i en sekvensiell rekkefølge.



DoubleStory

Bens Story	Arthurs Story
1 Ben: I don't understand this at all! Didn't you say that this was a serious book about culture? That it *really* says what's going on?	**1 Arthur**: Yes, I did.
2 Ben: What is this story doing here then?	**2 Arthur**: That story is a parable. Mrs. Adelaide's house is a metaphor for modern culture.
3 Ben: So the author tells us what is *really* going on by telling us a story?	**3 Arthur**: The main message of the book is that we are, in effect, living in a house with failing foundations. That 'house' is our culture.
4 Ben: What do you mean by that? How can a culture be like a house with failing foundations?	**4 Arthur**: Well, to see how you really need to read the book. The whole book is a *justification* or a proof of that message. The analogy between the house with failing foundations and the culture is shown point by point.
5 Ben: What does that mean, that main message?	**5 Arthur**: A house with failing foundations is not a place where one can live. It is unsafe. Doing repaires is also meaningless. It is necessary to rebuild the foundations.
6 Ben: All right, all right, I know about the house, but what is the message about the *culture* ?	**6 Arthur**: It's the same: That our culture is not a safe place to live in. And that trying to fix problems or to do small improvements is no better than patching holes in a house with failing foundations.
7 Ben: That does sound pretty serious. I thought that story was a joke.	**7 Arthur**: It's not a joke. I am telling you that book is serious. You should not be deceived by its light-hearted style. I think that the author wanted to put sugar coating around his message, to make it easier to swallow. But the message itself is completely serious.
8 Ben: A bit *too* serious, for my taste. Is this perhaps another doomsday prophesy?	**8 Arthur**: No, not at all. The message of the book is, in fact, optimistic. I would even say *unusually* optimistic, considering that the book talks about the prospects of our culture.
9 Ben: Optimistic? Are you calling the idea of living in a house with failing foundation *optimistic* ?	**9 Arthur**: Yes, I am. To see why, think again about the metaphor: "The foundations are failing" is an identifiable situation which has a solution - reconstruction. We have all but given up trying to make a technological culture that can last. The book says it's possible. And a lot more than that: When a house is reconstructed, the result is usually not only safer, but also *better* ...
10 Ben: Wait a minute, Arthur. You are taking it for granted that the modern culture really *is* like a house with failing foundations, which I don't. I don't believe the story.	**10 Arthur**: You don't? Well, that is just excellent. Then you are exactly the kind of person this book has been written form.
11 Ben: What do you mean?	

Figur 7.3: Ben & Arthurs konversasjon som paralell-fortelling.

7.2 Lavnivåeksempler

7.2.1 Obdok Manual

Det tredje eksemplet viser en uhyre simplifisert og minimalistisk manual for dokumentspråket **Obdok**; eller snarere en kortfattet oversikt over byggesteinene i språket. Begrensing av informasjonsinnholdet er et bevist valg, for at dette ikke skal forstyrre eksempelets egentlige hensikt: å belyse bruk av `link`'er, *set-referanser* og *arv*, samt komposisjon av en mer komplisert dokumentstruktur enn de foregående eksempler. Manualen er bygget opp av fire `set` som både er relatert ved arv og ved referanser.

Legg også merke til forskjellen mellom deklarert og implisitt bruk av `link`-konstanter, i elementene *'dekl_link'* og subelementet *'link'* under *'forklaring'*.

rel.dk

```
high:

  <!-- 'manual'-definerer dokumentet som skal skrives:
  her en Obdok-mal
  'type' beskriver informasjontypen
  'innhold' peker til de evt. informasjonstyper denne
     informasjonstypen kan inneholde
  'forklaring' tekstlig forklaring
  -->
set manual{
    head mal;
    ingress type;
    array (set inftype) innhold;
    text forklaring;
    array (link dekl_link) relatert;
}

low:

  <!-- 'primitiv'-set beskriver primitive informasjonstyper
  i Obdok. set'et arver 'manual'
  'attributter' benyttes forattributtoppramsing
  -->
set (manual) primitiv{
     array (text att) attributter;
}


 <!-- peker'-set beskriver link og set-referanser
  i Obdok: arver 'primitiv'
  'to_delt' benyttes for å forklare hybridløsinig
  -->
set (manual) peker{
    ingress hybrid;
    text attributt;
}

  <!-- 'kompleks'-set beskriver komplekse informasjonstyper
  i Obdok: arver 'primitiv'
  'ikke_sykel' brukes for å ungå sykler i dokumentet
  -->
set (primitiv) kompleks{
    ingress ikke_sykel;
}
```

rel.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<obdok>
<high>
  <manual set_name="Obdok-Manual for nybegynnere!">
    <mal>Objektorientert Dokumentspråk (<b>Obdok</b>)</mal>
    <type>
      Objektorientert dokumentspråk som brukes til å
      komponere dokumenter på en ny og innovativ måte
    </type>
    <forklaring>
      Alle <b>Obdok</b>-dokumenter bygges opp av en samling
      <i>set</i>-elementer fordelt på høynivå- og lavnivåskop.
      Disse materialiseres igjennom en instansering i <b>
      <link src="http://www.w3.org">XML</link></b>-syntaks.
      <br/>
      <br/>
      Ved hjelp av <i>set</i>-konstruksjoner kan der også
      bygges <i>abstrakte informasjonstyper</i>. Mer om dette
      <link src="../Program/obdok.php?dekl=ait.dk&amp;inst=ait.xml">
      her!</link>
      <hr />
    </forklaring>

    <innhold>
      <inftype ref="set" />
    </innhold>
    <forklaring>Merk: siden er bare et eksempel og ingen komplett
      referanse over <b>Obdok</b>. Dette språket er i sin helhet
      beskrevet i  <link src="../Present/hvd.pdf">
      Objektorientert  Dokumentdesign i <b>XML</b></link>
      <!-- implisitt link -->
    </forklaring>
    <relatert array_name="Relaterte sider">
    <!-- Eksplisitt, deklarerte linke -->
      <dekl_link src="..//main.php"><b>Obdok</b>-site</dekl_link>
      <dekl_link src="http://www.w3.org"><b>W3C</b></dekl_link>
      <dekl_link src="../Present/hvd.pdf">
        Objektorientert  Dokumentdesign i <b>XML</b></dekl_link>
    </relatert>
   </manual>
</high>
<low>
  <kompleks set_name="set">
    <type>
       Kompleks, generell informasjonstype. Kan inneholde en
       homogen eller hetrogen mengde bestående av alle andre
       informasjonstyper:
    </type>
    <innhold array_name="Byggesteiner (informasjonstyper):">
      <inftype ref="array" />
      <inftype ref="text" />
      <inftype ref="ingress" />
      <inftype ref="head" />
      <inftype ref="formal" />
      <inftype ref="statement" />
      <inftype ref="picture" />
      <inftype ref="verse" />
      <inftype ref="set-referanse" />
      <inftype ref="link" />
    </innhold>
    <attributter array_name="Brukgergitte attributter:">
      <att>set_name</att>
      <att>ait</att>
      <att>ref</att>
    </attributter>
  </kompleks>

  <kompleks set_name="array" >
    <type>
      Kompleks men sekvensiell informasjonstype. Krever også av den
      inneholdte mengde at denne er homogen
    </type>
    <innhold array_name="Byggesteiner (informasjonstyper):">
      <inftype ref="text" />
      <inftype ref="ingress" />
      <inftype ref="head" />
      <inftype ref="formal" />
      <inftype ref="statement" />
      <inftype ref="picture" />
      <inftype ref="verse" />
      <inftype ref="set-referanse" />
      <inftype ref="link" />
```

```
    </innhold>
    <ikke_sykel>
      I tillegg til de nevnte informasjonstypene, kan array-konstanter
      også bygges opp av nye array-konstruksjoner.
    </ikke_sykel>
    <attributter array_name="Brukergitte attributter:">
      <att>array_name</att>
    </attributter>
  </kompleks>

<!-- Slutt på komplekse info-typer: primitive begynne -->

  <primitiv set_name="text">
    <type>
      Primitiv informasjonstype som representerer en ren,
      uformatert tekst.
    </type>
    <attributter array_name="Brukergitte attributter:" >
      <att>Ingen brukeroppgitte attributter</att>
    </attributter>
  </primitiv>

  <primitiv set_name="ingress">
    <type>
      Primitiv informasjonstype for mellom eller høynivås
      informasjonspresentasjon.
    </type>
    <attributter array_name="Brukergitte attributter:" >
      <att>Ingen brukeroppgitte attributter</att>
    </attributter>
  </primitiv>

   <primitiv set_name="head">
    <type>
      Primitiv informasjonstype på høyest nivå. Beregnet
      overskrifter, stikkord, etc.
    </type>
    <attributter array_name="Brukergitte attributter:" >
      <att>Ingen brukeroppgitte attributter</att>
    </attributter>
  </primitiv>

  <primitiv set_name="statement">
    <type>
      Primitiv informasjonstype som representerer et uttrykk,
      et sitat eller et utsagn som kommer fra en avsender.
      Avsenders navn o.l settes i <i>announcer</i>-attributtet.
    </type>
    <attributter array_name="Brukergitte attributter:" >
      <att>announcer</att>
    </attributter>
  </primitiv>

  <primitiv set_name="picture">
    <type>
      Primitiv informasjonstype med med referanse til et bilde
    </type>
    <forklaring>
      Elementrepresentasjonen av <i>picture</i>-informasjonstypen
      har fire attributter hvor av ett er påkrevet. Dette er
      <i>src</i>-attributtet.</forklaring>
    <attributter array_name="Brukergitte attributter:" >
      <att>src</att>
      <att>width</att>
      <att>height</att>
      <att>alt</att>
    </attributter>
  </primitiv>

  <primitiv set_name="formal">
    <type>
      Primitiv informasjonstype som  representerer lover,
      korollar, teorier, etc.
    </type>
    <attributter array_name="Brukergitte attributter:" >
      <att>form_name</att>
    </attributter>
  </primitiv>

  <primitiv set_name="verse">
    <type>
      Primitiv informasjonstype for fremstilling av lyriske
      strofer.
    </type>
```

```
    <attributter array_name="Brukgergitte attributter:" >
      <att>verse_name</att>
    </attributter>
  </primitiv>

<!-- Slutt på primitive info-typer: pekere begynne -->
  <peker set_name="set-referanse">
    <type>Pekertype for å referere til interne resurser </type>
    <attributt>
      Typen har et påkrevet attributt <i>set_name</i>-verdien
      til det refererte <i>set</i>
    </attributt>
  </peker>

  <peker set_name="link">
    <type>Pekertype for å referere til eksterne resurser </type>
    <attributt>Typen har et påkrevet attributt <i>src</i>
    som angir url-adresse til resursen
    </attributt>
    <forklaring>
      Kan både brukes eksplisitt og implisitt i dokumentet
    </forklaring>
  </peker>
</low>
</obdok>
```

Figur 7.4: Eksempelet viser relasjoner i Obdok, i tillegg til å være en manual for språket. Kun første del av eksempelet er vist i illustrasjonen

7.2.2 Eksempel med Abstrakte Informasjonstyper

Det siste eksemplet som vises er egentlig fire eksempler, representert i like mange `set`-konstanter. Hver av disse tar for seg en de fire abstrakte informasjonstypene. Tre av disse er alternative, polyskopiske modelleringer av temaer presentert tidligere i denne oppgaven. Det siste eksempelet er hentet fra Håvamål[1], og viser tre vers fra denne samling.

Eksemplenes hensikt er todelt. For det første skal de vise hvordan AIT'enes strukturelle komposisjoner fungerer i praksis. Videre er det ment å vise hvordan informasjonstypene fungerer i informasjonsformidlingen, som *«grensesnitt av en spesifikk komposisjon»*.

ait.dk

```
high:
set grapolog_test{
    set test_ref;
    ingress title;
}

low:
set  graphолog1{
  picture graph;

  array (head overskrift) overskrifter;
  array (text tekst) tekster;

}

set artikkel{
  head overskrift;
  ingress sub_title;
  text broed_tekst;
}

set ide1{
  formal form;
  array (text tekst) description;
}

set dikt{
  array(text add) noter;
  array (verse vers) diktet;
}
```

ait.xml

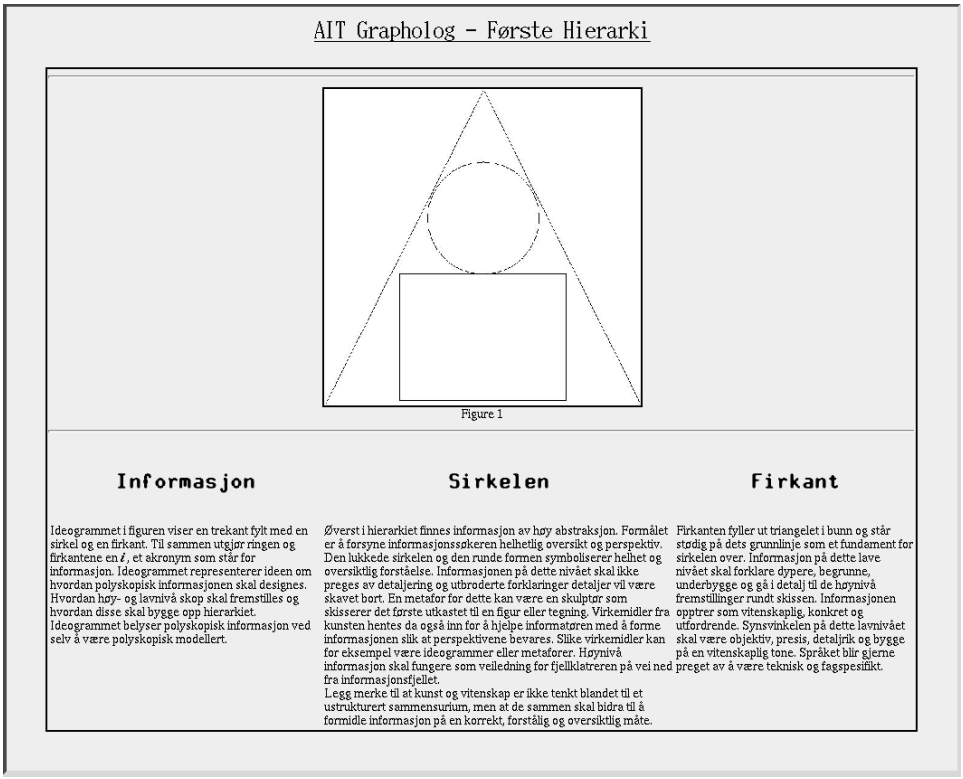
```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<obdok>
<high>

<grapolog_test set_name="Eksempel på ulike AIT-konstruksjoner">
  <title>
     Denne siden er kun ment som en simplifiserte eksempelside for ulike
     <i>Abstrakte Informasjonstyper</i> i <b>Obdok</b>
  </title>

  <test_ref ref="AIT Grapholog - Første Hierarki" />
  <test_ref ref="Column-AIT Eksempel" />
  <test_ref ref="Poetic-AIT Eksempel" />
  <test_ref ref="Ideonomy-AIT Eksempel" />
</grapolog_test>
```

[1] Hentet fra klippeslottet sin hjemmeside: `http://oaks.nvg.org/ma3ra4.html`

```
</high>

<low>

<graphолog1 set_name="AIT Grapholog - Første Hierarki" ait="grapholog">
  <graph src="http://folk.uio.no/hansogj/Hovedfag/Oppg/Present/triangel.jpg"
         height="350" width="350" alt="Figure 1" />
  <overskrifter array_name="First Hierarky Array Head">
    <overskrift>Informasjon</overskrift>
    <overskrift>Sirkelen</overskrift>
    <overskrift>Firkant</overskrift>
  </overskrifter>
  <tekster array_name="Første Hierarki">

    <tekst>Ideogrammet i figuren viser en trekant fylt med en sirkel
      og en firkant. Til sammen utgjør ringen og firkantene en
      <b><i>i</i></b>, et akronym som står for
      informasjon. Ideogrammet representerer ideen om hvordan
      polyskopisk informasjonen skal designes. Hvordan høy- og lavnivå
      skop skal fremstilles og hvordan disse skal bygge opp
      hierarkiet. Ideogrammet belyser polyskopisk informasjon ved selv
      å være polyskopisk modellert.
    </tekst>

    <tekst>Øverst i hierarkiet finnes informasjon av høy
      abstraksjon. Formålet er å forsyne informasjonssøkeren helhetlig
      oversikt og perspektiv. Den lukkede sirkelen og den runde formen
      symboliserer helhet og oversiktlig forståelse. Informasjonen på
      dette nivået skal ikke preges av detaljering og utbroderte
      forklaringer detaljer vil være skavet bort. En metafor for dette
      kan være en skulptør som skisserer det første utkastet til en
      figur eller tegning. Virkemidler fra kunsten hentes da også inn
      for å hjelpe informatøren med å forme informasjonen slik at
      perspektivene bevares. Slike virkemidler kan for eksempel være
      ideogrammer eller metaforer. Høynivå informasjon skal fungere
      som veiledning for fjellklatreren på vei ned fra
      informasjonsfjellet. <br />Legg merke til at kunst og vitenskap
      er ikke tenkt blandet til et ustrukturert sammensurium, men at
      de sammen skal bidra til å formidle informasjon på en korrekt,
      forståelig og oversiktlig måte.
    </tekst>
    <tekst>
      Firkanten fyller ut triangelet i bunn og står stødig på dets
      grunnlinje som et fundament for sirkelen over. Informasjon på
      dette lave nivået skal forklare dypere, begrunne, underbygge og
      gå i detalj til de høynivå fremstillinger rundt
      skissen. Informasjonen opptrer som vitenskaplig, konkret og
      utfordrende. Synsvinkelen på dette lavnivået skal være objektiv,
      presis, detaljrik og bygge på en vitenskaplig tone. Språket blir
      gjerne preget av å være teknisk og fagspesifikt.
    </tekst>

  </tekster>
</graphолog1>

<artikkel set_name="Column-AIT Eksempel" ait="column">
  <overskrift>Objektorientert Programmering</overskrift>
  <sub_title>Objekters evner til å arve andre objekter har vist seg
  som en av de viktigste årsakene til objektorienterings posisjon innen
  programmering pr. i dag.  Dette belyses nærmere i denne notisen.
  </sub_title>

  <broed_tekst>Objektenes evne til å arve egenskaper fra andre
    objekt(er) har også vært en viktig medvirkende faktor for å oppnå
    kortere, mer lettlest og ikke minst mer elegant
    kode. Hovedprinsippet går ut på at subklasser arver egenskaper fra
    superklasser. I den virkelige verden finnes mange gjenstander/ting
    som deler flere felles trekk med hverandre, men som har enkelte
    individuelle forskjeller. Slike konstellasjoner representeres
    gjerne i objektorienterte språk ved å definere den generelle
    klassen først. Her deklareres alle egenskaper felles for
    klassesystemet. Deretter defineres underklasser av denne. Disse
    arver superklassens egenskaper, men spesialiseres etter deres
    særegne behov. Et eksempel kan være gruppering av forskjellige
    kjøretøy. Superklasse ville kunne være 'kjøretøy', mens
    subklassene kunne være 'lastebil', 'personbil' og
    'motorsykkel'. Lastebilklassen kan igjen være superklassene for
    'vogntog' og 'trailer', personbil for 'varebil', 'cabriolet' og
    'sedan' , og motorsykkel kan ha subklassene 'touring', 'racing' og
    'chopper'. Det hele dreier seg altså om klassifisering av
    gjenstander hvilke gjenstand som skal i hvilke bås, og hvilke
    særegenskaper objektene i hver bås skal få.<br /> På tilsvarende
    måte som at objekter arver egenskaper, arver de også
    funksjonalitet gjennom de funksjonene som er deklarert i
```

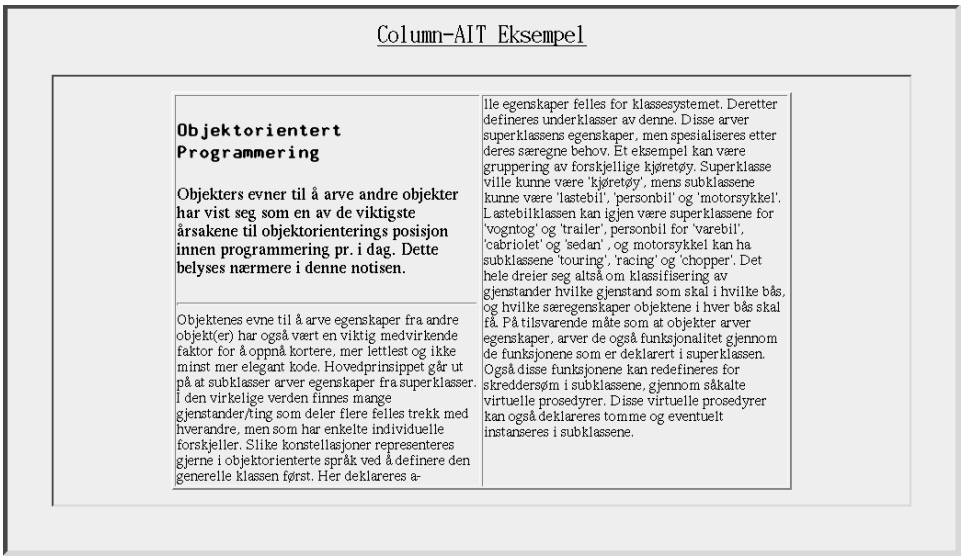
```
    superklassen. Også disse funksjonene kan redefineres for
    skreddersøm i subklassene, gjennom såkalte virtuelle
    prosedyrer. Disse virtuelle prosedyrer kan også deklareres tomme
    og eventuelt instanseres i subklassene.
  </broed_tekst>
</artikkel>

<dikt set_name="Poetic-AIT Eksempel" ait="poetic">
  <diktet array_name="Håvamål">
    <vers verse_name="6">
      Av eigen klokskap treng ingen skryte, men halde tanken i taume.<br />
      Du kjem ikkje brått i beit når du driv [1] gløgg og nesten teiande [2] i gardane.
    </vers>
    <vers verse_name="10">
      Du ber ikkje betre bør i bakken enn bra med vett.<br />
      Det er betre enn gull i framand gard; godt vett er stakkars trøst [1].
    </vers>
    <vers verse_name="64">
      Ein mann med vett skal nytte makta si med lempe og som det passar.<br />
      Kjem han saman med tigrar [1], finn han støtt at ingen er djerv framfor<br />
      alle.
    </vers>
  </diktet>

  <noter array_name="Tillegg til Hovamål">
    <add>
      <b>Add 6:</b>
      1. Drive (her): svive, vanke, fare (omkring) o.a.
      2. Eller mest teiande - eig. tagal: (1) som taler lite; (2) teiande.
    </add>
    <add>
      <b>Add 10:</b>
      1. Også lindring, tillit.
    </add>
    <add>
      <b>Add 64:</b>
      1. "Tigrar" er ein metafor for "djerve"
      her. Hamskiftar-freisting: "I lag med tigrar er ingen jerv
      framfor andre". (Humor)
    </add>
  </noter>
</dikt>

<ide1 set_name="Ideonomy-AIT Eksempel" ait="ideonomy">
  <form form_name="Funksjonalitet">
    defineres i <b>Obdok</b>-sammenheng til å være kommunikasjon
    av informasjon
  </form>
  <description>
    <tekst>
       Et <b>Obdok</b>-dokument er en strukturert sammenstilling av
byggesteiner med et formål å formidle innhold.
<i>Funksjonaliteten</i> til et slikt dokument vil dermed være
kommunikasjon av dette innholde --- altså informasjon
      </tekst>
      <tekst>
        Å programmere en datamaskin handler om manipulering av data
--- altså databehandling. Som beskrevet  vil et programs
funksjonalitet tilsvare summen av de opperasjoner og kall som
utføres på de datatyper og funksjoner programmet består av, og
at dette er en hierarkisk modell.  Et kjørende program er
således i konstant dynamisk bevegelse og det er denne
helhetlige bevegelse som gjør programmet.
      </tekst>
      <tekst>
        Informasjonsbærende dokumenter er på sin side statiske
konstanter, likeledes som at deres byggesteiner er statiske og
konstante. Til tross for denne stillestående natur, åpner
definisjonen opp for et funksjonsbegrep for
<b>Obdok</b>-dokumenter. Dette funksjonsbegrepet sier at
dokument har funksjonalitet når det fremstilles for en leser
(eller annen mottaker).  Denne kommunikasjonen opptrer som en
dynamisk handling --- i det leser mottar og tolker
informasjon. Til forskjell fra dataprogrammer er det altså
ikke manipulasjon av dokumentets byggesteiner som avgrenser
dokumentets funksjonalitet, men manipulasjon av
leseren<note>Merk: å manipulere leser ikke nødvendigvis å føre
bak lyset, men å påvirke</note>. Ved å ha innført et
kvantifisert informasjonsbegrep, er det også mulig å si at den
samlede funksjonalitet til et objektorientert dokument, er lik
summen av funksjonaliteten til byggesteinene dokumentet er
satt sammen av. En modulær inndeling av informasjonsdokumenter
medfører således også en modulær inndeling av funksjonaliteten
som inntar samme (hierarkiske) struktur som dokumentet. Dette
```

```
tilsvarer programmers funksjonalitet.
      </tekst>
</description>

</ide1>

</low>
</obdok>
```

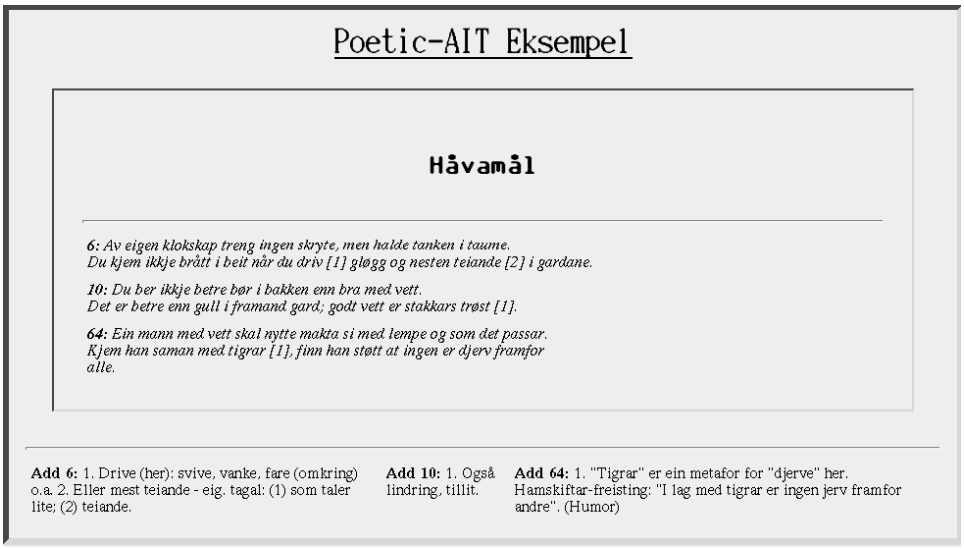


Figur 7.5: *Grapholog*-eksempel

Figur 7.6: *Ideonomy*-eksempel



Figur 7.7: *Coulmn*-eksempel



Figur 7.8: *Poetic*-eksempel

Kapittel 8

Avslutning

8.1 Oppsummering

8.1.1 Problemstilling

Med utgangspunkt i et stadig økende behov for mer raffinerte metoder for behandling av informasjon, argumenteres det i denne oppgaven for å føre en analogi fra objektorientert programmering til design av informasjon. Det objektorienterte programmeringsparadigmet har gjennom nærmere et halvt århundres bruk bevist sin fordelaktige beskaffenhet for strukturering av programkode. Formidling av informasjon har derimot et preg av å være umoderne, en tradisjonell egenart som nedstammer fra Guthenbergs bokpresse og oral formidlingsteknikk. Informasjonsdesign befinner seg således mye i samme situasjon som informatikken gjorde før objektorienteringen. Siden informasjonens strukturelle egenskaper vil få en stadig mer sentral rolle i tiden, er det derfor naturlig å forsøke gjøre de samme grep med informeringen, som objektorientering gjorde med programmeringen. Samtidig ble to nyvinninger tilgjengelige i Polyskopisk Modellering (Karabeg 2001*a*) og **XML**-baserte teknologier (XML n.d.) . Disse kunne legges til basis for å føre en analogi mellom objektorientering og dokumentdesign. Veien lå således åpen for å designe det objektorienterte dokumentspråket **Obdok**.

8.1.2 Obdok

Gjennom å studere og å bestemme de særegne kjennetegnene til de objektorienterte programmeringsspråkene, for så å forsøke å finne en måte å representere disse på i **XML**, er **Obdok** blitt modellert som et dokumentspråk for komponering av objektorienterte dokumenter. Til grunn

for modelleringen har i tillegg teorien i fra Polyskopisk Modellering ligget som et rammeverk for utformingen. Språket har en "kvantifisert" forestilling av informasjon, en oppfatning som stammer fra *infon*-begrepet (Hagen et al. 2000). I **Obdok** er disse enhetene gitt navnet *informasjonsenheter* og opptrer som forskjellige *informasjonstyper* representert av ulike **XML**-elementer. Informasjonstypene reflekterer programmeringsspråkenes *datatyper*. Det går et skille mellom *primitive* og *komplekse* informasjonstyper. Generelt kan det sies at primitive informasjonstyper *ikke* kan inneholde underordnede elementstrukturer, i motsetning de komplekse informasjonstypne.

Sentralt for komponering av objektorienterte dokumenter står den komplekse informasjonstypen `set`. Dette er en heterogen mengde som både er strukturell og kontekstuell ramme for de øvrige informasjonstypene. Dette kan sammenlignes med programmeringsspråkenes objekt. Innføring av de forskjellige informasjonstypene har dessuten muliggjort at analogien kan følges tett også hva de overordnede struktureringsegenskaper gjelder. *Arv* og *abstrakte datatyper* i informatikken er således blitt til *arv* og *abstrakte informasjonstyper* i **Obdok**. Forøvrig ble det valgt en hybrid løsing for resurslinking. Denne *peker*-funksjonaliteten skiller mellom interne og eksterne resurser.

Fundamentet til dette språket er lagt i en dokumenttype-definisjon. Denne definerer alle informasjonstypene/pekerne og bestemmer hva slags mulig innhold disse kan ha. Relatert til **DTD**'en er en deklarasjonssyntaks som er utformet til **Obdok**. Denne syntaksen gir bruker mulighet til å sette opp dokumentstrukturen i en objektorientert omgivelse, isteden for en forfatter- eller markup-omgivelse. Syntaksen åpner dessuten for at byggesteinenes navngiving og komposisjon også er brukers ansvar. Materialiseringen av denne strukturen og disse byggesteinene, skjer altså gjennom **XML**-elementer i et **XML**-dokument, som blir kalt instanseringsfil. Instansering holdes adskilt fra deklarasjonen gjennom egne deklarasjonsfiler.

Obdok-deklarasjoner og instanser prosesseres gjennom et pythonskript (*obdok.py*) for å konvertere og smelte disse sammen til en struktur av standard **XML**-format. Denne strukturen kan så visualiseres gjennom en **XSLT**-transformasjon (*obdok.xslt*).

8.2 Konklusjon

Resultatet av dette arbeidet er blitt et språk som skal kunne bistå informasjonsformidling, både under utarbeidelse, så vel som under konsumering av informasjonen. Hvor godt språket møter sin intensjon, og

hvilke bidrag språket virkelig yter, er nødvendige spørsmål å diskutere etter endt arbeid.

8.2.1 Styrker og Svakheter med Obdok

Der finnes selvfølgelig en rekke svakheter ved et prototypearbeid som dette. Mange av disse må ansees som (teknisk) enkle, men tidkrevende å utbedre. Eksempler på dette er utbedring av stilsett og av konverterens feilmeldinger. Andre egenskaper ved **Obdok** gir dog større utslag på hvor godt språket utfyller sin hensikt, og blir således diskutert grundigere her:

Brukervennlighet Språket er ikke testet ut i bruk i andre sammenhenger enn eksemplene fra kapittel 7, og den eneste som har benyttet seg av språket er dets opphavsmann. Det vil derfor være vanskelig å finne fullgode svar på hvor brukervennlig **Obdok** er modellert, før språket (eventuelt) blir tatt i bruk i større målestokk. Derimot gir erfaringer fra utarbeidelsen av nevnte eksempler et inntrykk av visse egenskaper ved språket. Blant annet kan det sies at:

1. For de fleste informasjonstyper er både deklarasjon og instansering handlinger som følger nokså intuitivt fra erfaring med objektorienterte programmeringsspråk og arbeid med **HTML**/**XML**.
2. For både deklarasjons- og instanseringsfilen er håndtering av struktur også i tråd med tidligere arbeid innen programmering og dokumentskriving i **HTML**/**XML**.
3. At språket benytter både #PCDATA og attributter for innkapsling av informasjon kan virke forvirrende, særlig om en ikke har god kunnskap til språkets oppbygging og konverterens funksjonalitet. Dette problemet skriver seg helt tilbake til **XML**-syntaksen, som jo benyttes for instansering av de objektorienterte dokumentene.
4. En hybrid pekerløsning kan virke forvirrende og unødvendig. `link`-informasjonstypen er lett å forstå, i all sin likhet til **HTML**s *anchor*-element. *set-referanser*, derimot, er noe mer komplisert, siden dette har samme elementnavn som den komplekse informasjonstypen `set`.

Informasjonstypeutvalget De informasjonstypene som er valgt ut til å representere informasjonsenhetene er intet endelig eller absolutt utvalg. Det kan således ligge mye arbeid i utprøving av deres egenskaper, og i

utbedring av utvalget. Særlig er dette gjeldende for de abstrakte informasjonstypene, presentert i kapittel 4.3.3. Dette fordi den grunnleggende teori (Polyskopisk Modellering) kun gir retningslinjer for design av informasjon. Teorien definerer dermed ikke noe endelig utvalg av bestemte skop. Da `set`-informasjonstypen er relatert til skopene ved at hvert `set` fungerer som det kontekstuelle rammeverk i **Obdok** (se kap. 4.1.1 på side 36), og de abstrakte informasjonstypene er predefinerte mønstre for oppbygging av slike `set`, finnes heller ingen endelig mengde abstrakte informasjonstyper. Det finnes sogar ingen begrensning for komposisjon av `set`-konstanter. Siden ikke *infon*-begrepet, som informasjonstypene bygger på, er klassifisert og inndelt i en bestemt mengde, er heller ikke utvalget av informasjonstypene endelig.

En måte å løse problemet med begrenset utvalg av informasjonstyper (både konkrete og abstrakte), er å legge denne spesifiseringen til bruker. En slik utvidelse av språket vil føre de objektorienterte dokumentspråkene inn i en ny generasjon, eller til et nytt nivå. Dette kan igjen sammenlignes med **XML**'s rolle som metataggspråk i forhold til tidligere markupspråk. Problemet med en slik løsning er å finne en måte å kombinere brukerdefinerte informasjonstyper med strukturell dokumentkomposisjon.

Predefinerte AIT'er Løsningen som benyttes for å angi at et `set` er av en bestemt abstrakt informasjonstype, kan optimaliseres. En AIT deklareres som et ordinært `set`, og det er opp til bruker å passe på at denne representasjonen sammenfaller AIT'ens komposisjon. **XML**-representasjonen må så følge denne komposisjonen med korrekte elementnavn og korrekt struktur. En alternativ løsning til dette, ville være å deklarere et `set` av en bestemt abstrakt informasjonstype, eksempelvis *'grapholog'*. Instanseringen av denne kan foregå implisitt, noe som vil si at informasjonstypene benyttes uten deklarasjon. **XML**-representasjonen vil med andre ord bestå av elementer som bærer informasjonstypes navn, istedenfor navn bestemt av bruker i deklarasjonen. En slik løsning forsterker dessuten oppfattelsen av det predefinerte og spesifikke i komposisjonen, og motvirker følelsen av det "tilfeldige".

For å klargjøre dette, vises eksempelet fra kapittel 7.2.2 gjennom en slik alternativ løsning (eksempelet viser kun gjelden konstant):

Deklarasjon:

```
low:
  set triangle(ait:grapholog){
  }
```

Instansering:

```
<triangle set_name="AIT Grapholog - Første Hierarki" ait="grapholog">
  <graph src="triangel.jpg" height="350" width="350" alt="Figure 1" />
  <array array_name="First Hierarky --- Array Head">
    <head>Informasjon</head>
    <head>Sirkelen</head>
    <head>Firkant</head>
  </array>
  <array array_name="Første Hierarki">
    <text>Ideogrammet... </text>
    <text>Øverst... </text>
    <text>Firkanten... </text>
  </array>
</triangle>
```

Obdok Konverteren Første del av konverteren har mange felles egenskaper med tradisjonelle programkompilatorer. Disse egenskapene ble bare *delvis* utnyttet som følge av manglende kunnskaper om kompilatorteknikk. En del effektivisering og utbedring kunne helt sikker ha blitt gjort om viten fra dette fagfeltet hadde blitt utnyttet i større grad.

8.2.2 Svar til Forventning

Det argumenters i denne oppgaven for at tradisjonelle dokumentspråk, eksempelvis **HTML**, binder brukeren til tradisjonelle, statiske og ineffektive dokumentstrukturer. For å modernisere og effektivisere informasjonsprosessen, hevdes det at informasjonsdesigneren må komponere dokumentstrukturer frigjort fra denne tradisjon. Dette vil kun være mulig med et verktøy som støtter et slikt brudd. **XML** er nettopp et slikt verktøy; et språk som tillater bruker å bygge sine egne dokumentkomposisjoner. **XML** er altså et verktøy for å definere dokumenters syntaks. I tillegg innehar språket en definert syntaks for instansering av disse komposisjonene. **XML** fungerer således som en relasjon mellom et høynivå dokumentspråk og en lavnivås syntaktisk beskrivelse. Sammenlignes dette med informatikken finner vi at **XML** i så måte tar assemblykodens plass som bindeledd mellom strukturerte programmeringsspråk og maskinkode.

På hvilke måte kan det så sies at **Obdok** oppfyller sine intensjoner som et nytt dokumentformat for strukturering av informasjon på en innovativ og rasjonell måte. Språket har, ved hjelp av **XML**, mestret å implementere en anvendelig syntaks for fremstilling av polyskopisk informasjon. **Obdok** bygger også på egenskaper fra objektorienterte programmeringsspråk, samt kunnskaper fra arbeid med slike; egenskaper og erfaringer som også er implantert inn i språket. **Obdok** innfører således

en objektorientert tankegang innen behandling og formidling av informasjon. Dette er med andre ord den samme strukturelle omlegging av dokumentspråkene, som objektorienterte språk gjorde med programmeringskoden på sekstitallet. **Obdok** tilbyr med dette både komponist og konsument en strukturert tilnærming til informasjonen innenfor et polyskopisk rammeverk. Dette er en tilnærming som er relatert til måten mennesker erverver seg kunnskap om objekter i den naturlige verden. En tilnærming som skal forbedre og effektivisere formidling av informasjon (Karabeg 2000*b*).

Komposisjon av dokumentstruktur til tross; det er viktig å huske at selv om **Obdok** tilbyr brukere en ny og innovativ måte å gjøre dette på, baseres fremdeles det kontekstuelle innhold på naturlige språk. Uavhengig av miljø og arbeidsrammer vil derfor fremdeles mye av informasjonens beskaffenhet bero på forfatterens språklige evner og kunnskaper. **Obdok**s rolle vil bare kunne være et redskap til støtte for komponisten i under utarbeidelse, og konsument under tilegning, av slik informasjonen.

8.2.3 Erfaringer med Støtteverktøy

Til grunn for arbeidet med **Obdok** har det hele tiden befunnet seg en nysgjerrighet og trang til å prøve ut nye teknologier — da særskilt **XML**-baserte teknologier. Erfaringene jeg sitter igjen med er ikke utelukkende positive.

XML er et velegnet språk for definisjon av dokumentkomposisjoner. Syntaktisk er språket lett å lære, intuitivt å forstå og enkelt å implementere — problemet er å lære støtteverktøy som **Xpath** og **XSLT** (Bray n.d.*a*). Applikasjoner og prosesseringer fanger også lett opp syntaksen. Dette gjør at språket er meget velegnet for generering av kode. At språket er tilknyttet en dokumenttype-definisjon gjør også språket mer anvendelig for gjenbruk, flerbruk og til hjelp med rettledning og kontroll av **XML**-strukturen. At språket er plattformsuavhengig er også en fordel når en arbeider med spredningsdokumenter (eng: shared documents).

Ulempene med språket er for det første at uferdige spesifikasjoner og mangel på implementasjoner har ført til at **Obdok**, som jo er instansert i **XML**, må *reduseres* til **HTML**-dokumenter. Redusert fordi **HTML** er et subsett av **XML**. Når koden ikke genereres automatisk, vil det derfor fremdeles i mange sammenhenger være like effektive å benytte **HTML**. At **XML** holder struktur og presentasjon adskilt, er heller ikke unikt for dette språket. Derimot er dette også gjeldende for **HTML 4.0** & **CSS**. **HTML** kan altså også benyttes ved fremstilling av informasjonen med forskjellige visuelle presentasjoner.

At **XML**-strukturer innehar egenskapen av å både kunne tolkes som sekvensielle hendelser *og* som en trestruktur, er til i mange tilfeller til stor fordel under prosessering av dokumenter. Ulempen med dette, er at mens **SAX**-parseren er effektive og resursbesparende, mangler denne manipulasjonsmuligheter som finnes i **DOM**-parseren. Sistnevnte er på sin side verken effektive eller resursbesparende. Tvert i mot; etter parsering blir **DOM**-treet i sin helhet lagret i maskinens minne. Siden **XML** er et ordrikt (verbost) språk, blir altså datamengden lett stor. Regulære uttrykk er i så måte en like så effektiv behandlingsmåte av mange **XML**-strukturer (Bray n.d.*b*).

Skillet mellom bruk av attributter og tekstlig innhold kan lett virke forvirrende. Det er også et spørsmål om *hvorfor* **XML** er blitt spesifisert med en slik delt løsning. Alt som blir representert av attributter i **XML** *kunne* ha blitt representert som elementsinnhold, adskilt fra elementets tekstlige innhold. På den andre siden blir enkelte tilfeller langt enklere og mye mer elegante ved bruk av attributter. Eksempelvis ville informasjonstypen `statement` fått en mye mer uheldig utforming ved å legge avsender (`announcer`-attributtet) som et eget element inne i informasjonstypen.

8.2.4 Fremtidig Arbeid

Der er særlig to videreutviklinger av **Obdok** som vil utbedre språket og gjøre det mer anvendelig for brukere. Dette er brukerdefinering av informasjonstyper, og det er en videre forskning på brukervennligheten til språket. Når det gjelder brukerdefinerte informasjonstyper, er dette omtalt tidligere i dette kapittelet, se avsnitt 8.2.1. Språkets brukervennlighet, derimot, er et felt som først og fremst må belyses gjennom en analyse og en brukerundersøkelse. Herunder ligger også utvikling av eventuelt andre løsninger når det gjelder nevnte problemer som deklarasjon av abstrakte informasjonstyper, men også egenskapene til pekerfunksjonaliteten. Selv om import av innhold med skjuling/vising fremstår som en mer raffinert måte å linke sammen resurser, som er teknisk mulig å gjennomføre også for kilder fra fjerne lokasjoner, vil heller ikke en slik løsning være det endelige svar på problemene rundt spagettilenking, redundant informasjonsmengde, eller døde linker. Endeløse *set-referanser*, eller inkluderte bolker, vil gi en like så ustrukturert, "spagettiformet" informasjonsmengde som den vi kjenner i dag. Et slikt grep vil altså ikke oppnå målsetningen om å gi informasjonen nødvendig perspektiv. Å finne alternative løsninger for hypertekst/hypermedia vil derfor være et like viktig arbeidsområde i fremtiden, som utarbeiding av frittstående dokumenters struktur.

Under utvikling av brukervennlighet faller naturligvis også forbedring av stilsett og presentasjon inn. Det stilsett som er laget til denne versjonen av **Obdok** er ment å ha et enkelt format for å rette fokus på de strukturelle egenskaper ved språket. Samtidig som dette fungerer for generell presentasjon, innebærer det forenklede og generelle at stilsettet ikke passer inn i skreddersydde prosjekter. Hvor mye arbeid som derfor er nødvendig å legge ned for å finne det "perfekte" stilsett er derfor et diskutabelt tema, om likevel fremtidige brukere vil utarbeide sine egne, spesialtilpassede presentasjoner av dokumentene.

Bibliografi

Bray, T. (n.d.*a*), Why xml doesn't suck. http://tbray.org/ongoing/When/200x/2003/03/24/XMLisOK.

Bray, T. (n.d.*b*), Xml is too hard for programmers. http://www.tbray.org/ongoing/When/200x/2003/03/16/XML-Prog.

Bush, V. (1945), 'As we may think', *The Atlantic Monthly* **176**(1), 101–108.

CSS (n.d.), *Cascading Style Sheetsw3:css*. http://www.w3.org/Style/CSS/.

DOM (n.d.), *Document Object Model (DOM)*. http://www.w3.org/DOM/.

Eriksen, T. H. (2001), *Øyeblikkets tyranni - rask og langsom tid i informasjonssamfunnet*, Aschehoug.

Goguen, J. A. (1997), Toward a social, ethical theory of infromation, *in* 'Social Science, Technnical Systems and Cooperative Work', Lawrence Erlbaum Associates, pp. 27 – 56.

Goguen, J. (1994), 'Requirements engineering as the reconciliation of technical and social issues', *Requirements Engineering: Social and Technical Issues* pp. 165 – 199. edited with Marina Jirotka.

Hagen, J. E., Amdal, G. & Nergaard, A. (2000), 'Beyond truth and falsehood'.

HTM (n.d.), *HyperText Markup Language*. http://www.w3.org/MarkUp/.

Jacobsen, I., Booch, G. & Rumbaugh, J. (1999), *The Unified Software Development Process*, Addison, Wesley, Longman Inc.

Karabeg, D. (1999), 'Information design challenge', (Institute for Informatics Report 280). http://www.ifi.uio.no/polyscope/id.shtml.

Karabeg, D. (2000*a*), Ideograms in polyscopic modeling, *in* 'Proceedings of the Infovision2000 conference, London, England'. http://www.ifi.uio.no/polyscope/id2000.pdf.

Karabeg, D. (2000*b*), User manuals with views, *in* 'presented at Expert Forum Manual Design, Eskilstuna, Sweden'. http://www.ifi.uio.no/polyscope/manuals-abstract.shtml.

Karabeg, D. (2001*a*), Information design by polyscopic modeling. Bokskisse.

Karabeg, D. (2001*b*), 'Polyscopic modeling – a general-purpose information design methodology', *Originally prepared for Information Design 2001 conference in Coventry, Great Britain* . http://www.ifi.uio.no/polyscope/Polymod.pdf.

Karabeg, D. (2002), 'Designing information design', *Information Design Journal 10(3)* . Lecture presented on the Designing Design conference, Oslo, October 2001.

Latour, B. (1987), 'Science in action', Open University Press.

Law, J. (1986), On the methods of long-distance control: Vessels, navigation and the portuguese route to india. published by the Centre for Science Studies and the Department of Sociology, Lancaster University at http://www.comp.lancs.ac.uk/sociology/soc077jl.html.

Linde, C. (2001), 'Narrative and social tacit knowledge', *Journal of Knowledge Management* **5**, 160 – 171. Emerald.

Lin (n.d.), *XML Linking Language (XLink) Version 1.0*. http://www.w3.org/TR/xlink/.

Lowe, D. & Hall, W. (1999), *Hypermedia & the Web - an engineering approach*, Wiley & Sons.

Lundeby, E. (1984), *Lexi, Nye Ord - Vanskelige Ord - Fremmedord*, Vol. 3. utg., Kunnskapsforlaget.

Mathiassen, L., Munk-Madsen, A., Nielsen, P. A. & Stage, J. (2000), *Object-oriented Analyses & Design*, Marko Publishing ApS, Aalborg, Denmark.

Maus, A. (n.d.), Objektorientert programmering og systemutvikling – en kortfattet innføring. kompendium IN-OBJ2-EVU: "Objektorientert systemutvikling med Java og UML", UiO.

Ole Hanseth, E. M. (1998), 'Understanding information infrastructure', http://www.ifi.uio.no/~oleha/Publications/bok.html. kapittel 6.

Orwel, G. (1949), *1984*, Martin and Warburg Ltd 1949.

Pat (n.d.), *XML Path Language (XPath)*. http://www.w3.org/TR/xpath/.

Pyt (n.d.), *Python*. http://www.python.org/.

Sab (n.d.), *Sablotron processor*. http://www.gingerall.com/charlie/ga/xml/p_sab.xml.

SAX (n.d.), *Simple API for XML*. http://www.saxproject.org/.

Taule, R. (1993), *Escolas Ordbok*, Escola Forlag.

Weiss, M. A. (1995), *Data Struktures and Algoritm Analysis*, The Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-9057-X.

Wurman, R. S. (2001), *Information Anxiety II*, QUE, Indianapolis, USA. ISBN: 0-7897-2410-3.

Xan (n.d.), *Xanadu*. http://www.xanadu.com/.

XML (n.d.), *Extensible Markup Language (XML) 1.0 (Second Edition)*. http://www.w3.org/TR/2000/REC-xml-20001006.

XSL (n.d.), *Extensible Stylesheet Language (XSL) Version 1.0*. http://www.w3.org/TR/xsl/.

XTR (n.d.), *Extensible Stylesheet Language Transformations (XSLT) Version 2.0*. http://www.w3.org/TR/xslt20/.

Tillegg A

Appendiks

A.1 Apendix

A.2 Obdok – DTD

```
<!-- verdiene til 'name' attributt som finnes i hvert element,
 stammer fra konstantnavnet i instanseringsfilen -->

<!DOCTYPE obdok [
<!-- obdoc er dokument-entiteten i disse dokumentene -->
    <!ELEMENT obdok (set)* >

<!-- Kosmetiske elememnter (som i HTML): begynner -->
    <!-- siden note oversettes i obdok.py, mens de andre kosmetiske
    elementene ikke prosesseres, vil ogsaa note-elementet trenge et
    name-att. -->
    <!ELEMENT i (#PCDATA | b |  br | hr )* >
    <!ELEMENT b (#PCDATA | i |  br | hr )* >
    <!ELEMENT note (#PCDATA | i |  br | hr )* >
    <!ATTLIST note name CDATA #IMPLIED>
    <!ELEMENT br EMPTY>
    <!ELEMENT hr EMPTY>
<!-- Kosmetiske elementer:slutter -->

<!-- primitive informasjonstyper: begynner -->
  <!-- text: begynner -->
    <!ELEMENT text (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST text name CDATA #REQUIRED>
  <!-- text: slutter -->

  <!-- ingress: begynner -->
    <!ELEMENT ingress (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST ingress name CDATA #REQUIRED>
  <!-- ingress: slutter -->

  <!-- head: begynner -->
    <!ELEMENT head (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST head name CDATA #REQUIRED>
  <!-- head: slutter -->

  <!-- picture: begynner -->
    <!ELEMENT picture EMPTY>
    <!ATTLIST picture src CDATA #REQUIRED
                      width CDATA #IMPLIED
      height CDATA #IMPLIED
      alt CDATA #IMPLIED
      name CDATA #REQUIRED>
  <!-- picture: slutter -->

  <!-- statement: begynner -->
    <!-- Et statemen skal inneholde et verbalt uttrykk (PCDATA)
    og avsluttereren av dette: announcer -->
```

```
    <!ELEMENT statement (#PCDATA | i | b | br | hr | link |  note)* >
    <!ATTLIST statement name CDATA #REQUIRED
                announcer CDATA #REQUIRED>
  <!-- statement: slutter -->

  <!-- verse: begynner -->
    <!-- Versets navn plasseres i verse_name -->
    <!ELEMENT verse (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST verse name CDATA #IMPLIED
                  verse_name CDATA #REQUIRED>
  <!-- verse: slutter -->

  <!-- novella: begynner -->
    <!-- Novellens navn legges i nov_name -->
    <!ELEMENT novella (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST novella name CDATA #REQUIRED
                   nov_name CDATA #REQUIRED>
  <!-- novella: slutter -->

  <!-- formal: begynner -->
    <!ELEMENT formal (#PCDATA | i | b | br | hr | link | note)* >
    <!ATTLIST formal name CDATA #IMPLIED
                  form_name CDATA #REQUIRED>
  <!-- formal: slutter -->

<!-- primitive informasjonstyper: slutters -->

<!-- komplekse informasjonstyper: begynner-->
  <!-- set: begynner -->
    <!-- set-elementet er det mest generelle elementet og kan
    inneholde andre sett og informasjonstyper. 'ID'-attributtet er
    identifikator, ait bestemmer om, og i saafall hvilke. abstrakt
    informasjonstype setet er, ref_from benevner hvilke evt. peker,
    setet stammer fra av og set_name navngir setet -->
    <!ELEMENT set ANY>
    <!ATTLIST set name CDATA #REQUIRED
                ID CDATA #REQUIRED
  ait CDATA #IMPLIED
  ref_from CDATA #IMPLIED
  ref CDATA #IMPLIED
            set_name CDATA #IMPLIED>
  <!-- set: slutter  -->

  <!-- array: begynner -->
    <!-- array_name er til evt. navngiving av listen/arrayen -->
    <!ELEMENT array ((array)* | (statement)* | (picture)* | (link)* |
                  (set)* | (ideonomy)* | (verse)*| (column)* |
     (grapholog)* | (poetic)* | (ingress)* | (text)* |
     (head)*) >
    <!ATTLIST array array_name CDATA #IMPLIED
                  name CDATA #REQUIRED>
  <!-- sequence: slutter-->
<!-- komplekse informasjonstyper: slutter -->

<!-- aitbegynner -->
  <!-- grapholog: begynner -->
  <!ELEMENT grapholog (picture, array, array)>
  <!ATTLIST grapholog name CDATA #REQUIRED>
  <!-- grapholog: slutter -->

  <!-- column: begynner -->
    <!ELEMENT column (head, ingress, text)>
    <!ATTLIST column name CDATA #REQUIRED>
  <!-- column: slutter -->

  <!-- ideonomy: begynner -->
    <!ELEMENT ideonomy (formal, array )>
    <!ATTLIST ideonomy name CDATA #REQUIRED>
  <!-- ideonomy: slutter -->

  <!-- poetic: begynner -->
    <!ELEMENT poetic (array, array)>
    <!ATTLIST poetic name CDATA #REQUIRED>
  <!-- ideonomy: slutter -->

<!-- abstract informationtypes: slutter -->

<!-- link : begynner -->
<!-- peker til ANDRE dokument -->
    <!ELEMENT link ANY >
    <!ATTLIST link src CDATA #REQUIRED
   name CDATA #REQUIRED>
<!-- link: slutter -->
```

```
]>
```

A.3 Konverter

```
import sys
import string
import re
from xml.dom.minidom import parse, parseString, getDOMImplementation

if len(sys.argv) < 3 or len(sys.argv) > 4:
    print sys.exit("invalid call - usage: parsering.py" ,
                   "deklareationfile.dkl instantiationfile.ins",
                   "( dtd-file.dtd )")


ID = 0 #Teller som holder rede på ID - økes med en pr nytt set
dekl=open(sys.argv[1])
dekl_text = dekl.read() #inneholder deklarasjonsfilen
dekl.close()


all_sets = {} #holder alle de forskjellige set-objektene som genereres
#under innlesning av deklarasjonsfilen
impl = getDOMImplementation()
utdoc = impl.createDocument(None, "obdok", None)
#Opretter et tomt <obdok />-element - som siden skal benyttes som
#document-element i treet vi skal prosessere

##################################################################
#                                                                #
#                     class set:                                 #
#                                                                #
##################################################################
class set:

    """Hvert set-deklarasjon representeres her med et set-objekt
    som inneholder en peker til EN evt. forelder, samt en hash over
    hvilke egenskaper settet kan inneholde - hashens nøkkel er
    konstantnavne og verdien peker til et infomasjonstype-objekt (et
    nytt set-obj, et array-obj eller et inftype-objekt)"""

    def __init__(self):
        self.ident = 'set'
        self.prop = {}#Initieres tom
        self.inherit = None
        #blir evt satt til å peke på et foreldre-set

    def set_inherits(self, inherits):
        self.inherit = inherits

    def get_inherits(self):
        return self.inherit

    def set_prop(self, key, value):
        self.prop[key]=value

    def get_prop(self, key):
        return self.prop[key]

    def get_ident(self):
        return self.ident
##################################################################
#                                                                #
#                      class inftype                             #
#                                                                #
##################################################################
class inftype:
    """Primitive informasjonstyper som text, head og picture
    lagres som inftype. Deres type lagers i type-variabelen
    """
    def __init__(self, type):
        self.ident = 'inftype'
        self.type = type

    def get_type(self):
        return self.type
```

```
    def get_ident(self):
        return self.ident
#####################################################################
#                                                                   #
#                     class array                                   #
#                                                                   #
#####################################################################
class array:
    """Representerer array-informasjonstypen. Prop.tuppelen tar
    vare på hvilke inftyper (inkl set / array) arrayet er bygget
    opp av - på samme måte som i set-objektene """

    def __init__(self):
        self.prop = {}
        self.ident = 'array'

    def set_prop(self, key, value):
        self.prop[key]=value

    def get_prop(self, key):
        return self.prop[key]

    def get_ident(self):
        return self.ident
#####################################################################
#                                                                   #
#                     def parse_set                                 #
#                                                                   #
#####################################################################
def parse_set(text):
    """Fjerner først alle kommentarer på formen /*blablabla*/
    Finner så første forekomst av 'set' - genererer en string
    som består av alle tegn fra og med 'set' til første forekomst
    av '}'"""
    regex = re.compile('\/\*(?:.(?!\*\/))*.\*\/', re.S)
    text = re.sub(regex, '', text)
    foo = 0 # teller ord i teksten
    txt = string.split(text)
    for word in txt:
        bar = foo
        if word == 'set':
            set_string = ''
            try:
                while  txt[bar] != '}':
                    set_string = string.join([set_string, txt[bar]])
                    bar = bar + 1

                create_set(set_string)
            except:
                print "<obdok><set set_name='ERROR' ID='0'><head>",
                print "Illegal construction:",
                print str(sys.exc_info()[1]),
                print "</head></set></obdok>"

        foo = foo + 1

#####################################################################
#                                                                   #
#                     def create_set                                #
#                                                                   #
#####################################################################
def create_set(text):
    """set_reg matcher alle linjer med 'set':
    0.element er (evt) super-settet,
    1.element er set-name
    2.element er set-deklarasjon
    """

    set_reg = 'set\s+(?:\((\w+)\)\s+)?(\w+)\s*{([^}]*)'

    set_match=re.findall(set_reg, text)
    for sm in set_match:
        name = sm[1]
        all_sets[name] = set()
        if sm[0] != "":
            all_sets[name].set_inherits(all_sets[sm[0]])
        #oppretter et nytt set-obj som heter sm[1]
        #Hvis klassen skal arve en klasse, ligger dennes navn i
        #sm[0]
        #Ellers er denne verdien tom ("")
        #resten av deklarasjonen av klassen finnes i sm[2](=dekl)
        for dekl in string.split(sm[2], ';'):
            if re.findall('\s+array', dekl):
                dekl_string = re.findall(
```

```
                '\s*array\s*\(\s*(.*)\s*\)\s*(\w*)', dekl)
            for all in dekl_string:
                #elem 0 = deklarasjon
                #elem 1 = array-navn
                arr = parse_array(all[0])
                all_sets[name].set_prop(all[1], arr)
        if re.findall('\s+set\s+', dekl):
            words = string.split(dekl, ' ')
            all_sets[name].set_prop(words[2], set())
            #oppretter et tomt set-objekt - objektet vil
            #siden initialiseres ved en algoritme som løper
            #gjennom all_sets og for alle pekere her - linke
            #disse opp mot andre set i all_sets hashen
        else:
            if not (re.findall('\s+array', dekl)
                    or re.findall('\s+set\s+',
                                  dekl)):
                words = string.split(dekl, ' ')
                if len(words) > 1:#første elem er alltid ' '
                    all_sets[name].set_prop(words[2],
                                            inftype(words[1]))
                #words[2] er konstantnavnet, words[1] er
                #inftypen oppretter et eget inftype-objekt
                #som hashtabbelen av denne nøkkelen peker på

################################################################
#                                                              #
#                     def parse_array                          #
#                                                              #
################################################################
def parse_array(text):

    """Gjennomløper en arrays innholdsfortegnelse for å finne ut
    hva slags inf.typer denne er bygget opp av. Hvis arrayet er
    bygget opp av substrukturer av flere arrayer, prosesseres disse
    rekursivt. Hvis arrayet er bygget opp av set-referanser,
    initieres disse tomme og fylles siden ut i metoden insert_ref """

    arr = array() #oppretter et tomt array-objekt
    if re.findall('\s*array\s*', text):
        split = string.split(text, ',')
        #en array av arrayer skilles  med kommas i deklarasjonen
        #dekl_list er nå på formen
        #['array(head hode) array1', 'array(head hode2) array2']
        for txt in split:
            dekl = re.findall('\s*array\s*\(\s*(.*)\s*\)\s*(\w*)',
                              txt)[0]
            arr.set_prop(dekl[1], parse_array(dekl[0]))
    else:
        words =  string.split(text)
        if words[0] == 'set':
            arr.set_prop(words[1], set())
        else:
            arr.set_prop(words[1], inftype(words[0]))
    return arr
################################################################
#                                                              #
#                     def apply_inherit                        #
#                                                              #
################################################################
def apply_inherit(set):
    """Kopierer forfederens prop-liste over i dette settets liste
    med andre ord - innfører arv
    """
    if set.get_inherits()!= None:
        ancestor = set.get_inherits()
        for p in ancestor.prop:
            set.prop[p] = ancestor.prop[p]
################################################################
#                                                              #
#                     def substitute                           #
#                                                              #
################################################################
def substitute(child):

    """Opretter ett nytt element i utdoc.strukturen. Alle
    set-elementer vil være søsken under enn så lenge. Set-referanser
    legges inn i set-elementet som et eget, tom element. Listen av set
    vil siden bli gjennomløpt og plasserer ut de elementer der de blir
    referert til (av de tomme set-referatene) """

    global ID
    if all_sets[child.tagName] == None:
        sys.exit("system error")
```

```
    else:
        foo = None
        #tomt element som siden skal plasseres i elem-elementet
        hold = all_sets[child.tagName]
        elem = utdoc.createElement('set')
        elem.setAttribute("set_name",child.getAttribute("set_name"))
        elem.setAttribute("ID", str(ID))
        ID = ID + 1
        if child.getAttribute('ait') != None:
            elem.setAttribute('ait', child.getAttribute("ait"))

        elem.setAttribute("name", child.nodeName)
        for children in child.childNodes:
            if children.nodeType == 1:
                # er kun interisert i ELEMENT_NODE
                try:
                    foo = sub_build(hold.get_prop(children.nodeName),
                                    children)
                except:
                    print "<obdok><set set_name='ERROR' ID='0'>",
                    print "<head>Inconsequense in constant name:",
                    print children.nodeName,
                    print str(sys.exc_info()[1]),
                    print "</head></set></obdok>"
                    sys.exit(1)
                elem.appendChild(foo)
    return elem

################################################################
#                                                              #
#                      def sub_build                           #
#                                                              #
################################################################
def sub_build(object, elem):
    """Bygger opp sub-strukturene av set-elementet -
    kalles rekursivt hvis i array-strukturene"""
    global ID
    foo = None # Tomt element som siden skal returneres
    if object.get_ident() == 'set':
        foo = utdoc.createElement('set')
        foo.setAttribute("name", elem.nodeName)
        foo.setAttribute("ref", elem.getAttribute('ref'))
        foo.setAttribute("ID", str(ID))
        ID = ID + 1
        #Oppretter kun et tomt element som siden skal fylles
        #med data/elementer når jeg i slutten av programmet
        #skal sørge for å oppfylle strukturen på dokumentet
        #allokerer mao bare plass til setet, settet
        #initialiserers i en annen del av dokuemntet
    if object.get_ident() == 'inftype':
        try:
            foo = utdoc.createElement(object.get_type())
            foo.setAttribute('name', elem.nodeName)
            for all in elem.childNodes:
                fo = all.cloneNode(1)
                #kloner noden og dens substruktur slik at
                #<br> <i> og <hr> ol blir med
                foo.appendChild(fo)
        except:
            print "<obdok><set set_name='ERROR' ID='0'>",
            print "<head>Inconsequense in constant name:",
            print elem.nodeName,
            print str(sys.exc_info()[1]),
            print "</head></set></obdok>"
            sys.exit(1)


    if object.get_ident() == 'array':
        foo = utdoc.createElement(object.get_ident())
        foo.setAttribute('name', elem.nodeName)
        for child in elem.childNodes:
            if child.nodeType == 1: #ELEMENT_NODE
                foo.appendChild(sub_build(
                    object.get_prop(child.nodeName), child))
                #Rekursiv gjennomløpning av alle set og
                #arrayer dette arrayet måtte være bygget opp av
    if elem.attributes != None:
        i=0
        while i < elem.attributes.length:
            #løper gjennom alle atributtene og kompierer disse
            #"rått" inn i det nye dokumentet
            pyle=elem.attributes.item(i).name
            foo.setAttribute(elem.attributes.item(i).name,
                             elem.getAttribute(pyle))
```

```
            i=i+1
    return foo
#####################################################################
#                                                                   #
#                       def insert_ref                              #
#                                                                   #
#####################################################################
def insert_ref(anchor_point, name):

    """ anchor_point forteller i hvilke element set-ref opptrer - dvs
    i hvilke element man skal plasserse sub-setet.  name gir
    key-verdien fra anchor_point sin all_sets eller .prop og kan
    fortelle oss om referansen opptrer i et set eller i en array

    For hvert set i all_sets finnes de setene som har set-referanser i
    sin prop-liste. set-referansene ble som vi husker satt tomme i
    initialisering i create_set / parse_Array. For hver
    array-forekomst må man søke rekursivt i denne for å finne evt
    instanser av set's i denne arrayens oppbygning

    Når man har funnet de tomme set-referansene finner man de
    tilsvarende set i elems-listen, referansens klone legges
    ankerpunktet hvor referasen opptrer, og det originale
    referanse-setet slettes fra set-listen """

    ref_set = [] #Lager en liste av refrete set, i tilfelle
                 #anchor_point inneholder mer enn en referanse
    insert_elem = None

    list = utdoc.getElementsByTagName(anchor_point.get_ident())

    for p in anchor_point.prop:
        if anchor_point.get_prop(p).get_ident() == 'array':
            insert_ref(anchor_point.get_prop(p), p)
        if anchor_point.get_prop(p).get_ident() == 'set':
            for elem in utdoc.getElementsByTagName('set'):
                if elem.getAttribute('name') == p:
                    for e in utdoc.getElementsByTagName('set'):
                        if (e.getAttribute('set_name')
                            == elem.getAttribute('ref')):
                            ref_set.append(e)

            for item in list:
                if item.getAttribute('name') == name:
                    insert_elem = item
                    #insert_elem finner i hvilke element
                    #ref_set skal plasseres
                    i = 0
                    if ref_set != [] and insert_elem != None:
                        children = insert_elem.childNodes

                        for child in children:
                            for refs in ref_set:
                                c_ref = children[i].getAttribute('ref')
                                r_setname = refs.getAttribute('set_name')
                                if c_ref == r_setname:
                                    ref_tmp = children[i].getAttribute('name')
                                    children[i]=refs.cloneNode(1)
                                    children[i].setAttribute('ref_from',
                                                             ref_tmp)
                            i=i+1


#####################################################################
#                                                                   #
#                      def remove_set                               #
#                                                                   #
#####################################################################
def remove_set(set):
    """Sletter de set som opptrer i et annet set via en ref_set
    all_sets hashen, da alle set som ligger på denne sett
    """
    tmp = None #Holder på node med ref = set-name
    sets = utdoc.documentElement.childNodes
    for p in set.prop:
        if set.get_prop(p).get_ident() == 'array':
            remove_set(set.get_prop(p))
        if set.get_prop(p).get_ident() == 'set':
            for elem in utdoc.getElementsByTagName('set'):
                if elem.getAttribute('ref_from') == p:
                    tmp = elem
                if tmp != None:
                    for all in sets:
                        if (all.getAttribute('name')
```

```
                            == tmp.getAttribute('name')):
                            sets.remove(all)

##################################################################
#                                                                #
#                      def fjern_norsked                         #
#                                                                #
##################################################################
def remove_nor(text):
    """Bytter ut norske bokstaver med konstellasjoner der er
    lite sansynlig man jeg en en tekst - disse konstellasjonene
    byttes tilbake i obdok.php
    """
    text = string.replace(text, 'Å', 'Aring;')
    text = string.replace(text, 'å', 'aring;')
    text = string.replace(text, 'Æ', 'AElig;')
    text = string.replace(text, 'æ', 'aelig;')
    text = string.replace(text, 'Ø', 'Oslash;')
    text = string.replace(text, 'ø', 'oslash;')
    return text

##################################################################
#                                                                #
#                      slutt på deklarasjoner                    #
#                                                                #
##################################################################

""" her begynner kjøringen av selve programmet"""

intxt=open(sys.argv[2])
try:
    indoc = parseString(remove_nor(intxt.read()))
#Leser inn instanseringsfil, og fjerner alle norske bokstaver
except:
    print "<obdok><set set_name='ERROR' ID='0'>"
    print "<head>mismatched tag:",
    print str(sys.exc_info()[1]),
    print "</unknown></head></set></obdok>"
    sys.exit(1)
intxt.close()
parse_set(dekl_text)

for set in all_sets:
    apply_inherit(all_sets[set])
    #Sørger for at et hvert set's prop-liste inneholder både de
    #prop som er deklarert for dette sett, samt de prop. som
    #er deklarert for dets super-sett. En redundantisk og dårlig
    #løsning, men en programmeringseffektiv løsning

##################################################################
#Utskriftprosedyre for feilsøk i programmet                      #
def utskrift(elem):                                             #
    for key in elem.prop:                                       #
        if elem.prop[key].get_ident() == 'inftype':             #
            print"%15s => %s" % (key,elem.prop[key].get_type())  #
        else:                                                   #
            print "%15s => %s" % (key,elem.prop[key].get_ident())#
            utskrift(elem.prop[key])                            #
#for all in all_sets:                                           #
#    print "Set : " +all +": " + all_sets[all].get_ident()      #
#    utskrift(all_sets[all])                                    #
#    print "-----------------------------------------------------"#
#    print "\n"                                                 #
#sys.exit(0)                                                    #
##################################################################


for child in indoc.getElementsByTagName('high')[0].childNodes:
    #Luker ut high/low elementene fra utdoc og og kaller på
    #substitute for alle hig/low sine set-subelementer
    if child.nodeType == 1:#ELEMENT_NODE har 1 i verdi
        utdoc.documentElement.appendChild(substitute(child))

if len(indoc.getElementsByTagName('low')) != 0:
    #Ikke påkrevet med low-sets
    for child in indoc.getElementsByTagName('low')[0].childNodes:
        if child.nodeType == 1:
            utdoc.documentElement.appendChild(substitute(child))

for set in all_sets: #bytter ut set-referanser med instanser av set
    insert_ref(all_sets[set], set)
for set in all_sets:
    remove_set(all_sets[set])
    #Fjerner de sett fra all_sets-hashen som også opptrer som
```

```
    #subset av andre set

sets = utdoc.getElementsByTagName('set')
ID = 0
for set in sets: #sørger for at alle set i utdoc har unike ID
    sets[ID].setAttribute('ID', str(ID))
    ID = ID+1

pretty_printed = utdoc.toprettyxml()
if len(sys.argv) == 4:

    dtd_file = open(sys.argv[3]).read()
    pretty_printed = string.replace(pretty_printed, '<obdok>',
                                    dtd_file+"<obdok>",)
print pretty_printed
```

A.4 Stilsett

A.4.1 Obdok – XSLT

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <head><title><xsl:value-of select="//@ID" /></title>
        <script type="text/javascript"
src="onclick.js" />
<link rel='stylesheet'  type="text/css" href="cstyle.css"/>
      </head>
      <body>
        <div class="center_head"><xsl:value-of select="//@set_name" /></div>
        <xsl:apply-templates  select="obdok/set"/>
      </body >
    </html>
  </xsl:template>
<!--onclick-funksjonen krever at nettleseren støtter
Document Object Model Level 2 HTML
http://www.w3.org/TR/2002/CR-DOM-Level-2-HTML-20020605/

Document Object Model Level 2 Style
http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113/
-->

  <xsl:template match="set">
    <div class="set">
      <div class="clickhead" >
        <xsl:attribute name="onclick" >swap_visibility('<xsl:value-of
     select="@ID" />')</xsl:attribute>
        <a class="hoover"><xsl:value-of select="@set_name" /></a>
      </div><!-- display:none -->
      <div style="display:none" ><xsl:attribute name="id"><xsl:value-of
           select="@ID" /></xsl:attribute>

<!-- tester på om set-et er en AIT som skal spesialprosesseres -->
<xsl:choose>
  <xsl:when test="@ait = 'grapholog'">
    <xsl:call-template name="grapholog"/>
  </xsl:when>
  <xsl:when test="@ait = 'ideonomy'">
    <xsl:call-template name="ideonomy"/>
  </xsl:when>
  <xsl:when test="@ait = 'column'">
    <xsl:call-template name="column"/>
  </xsl:when>
  <xsl:when test="@ait = 'poetic'">
    <xsl:call-template name="poetic"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:apply-templates />
  </xsl:otherwise>
</xsl:choose>
      </div>
```

```
    </div>
  </xsl:template>

<!-- array: begin -->

 <xsl:template match="array">
    <xsl:variable name="param" select="count(array)" />
    <xsl:call-template name="array-recursive" >
      <xsl:with-param name="inparam" select="$param" />
    </xsl:call-template>
  </xsl:template>

  <!-- array-recursive: begin -->
  <xsl:template name="array-recursive">
    <xsl:param name="inparam" />
    <xsl:variable name="vidde" select="100 div $inparam" />
    <!-- Deler inn hele bredden i 'inparam' like celler -->

    <p class="array">
      <p class="head"><xsl:value-of select="@array_name" /></p>

      <xsl:choose><!-- for multidimensjonale arrayer -->
        <xsl:when test="name(child::*) = 'array'">
  <table border="1"  cellspacing="2" cellpadding="2" width="100%">
    <tr valign="top">
    <xsl:for-each select="array">
      <td align="left">
      <xsl:attribute name="width">
         <xsl:value-of select="$vidde" />%
      </xsl:attribute>
        <xsl:variable name="param" select="count(array)" />
        <xsl:call-template name="array-recursive" >
          <xsl:with-param name="inparam" select="$param" />
        </xsl:call-template>
      </td>
    </xsl:for-each>
    </tr>
  </table>
 </xsl:when>
 <xsl:otherwise><!--for éndimensjonale arrayer -->
           <table border="0" cellspacing="2" cellpadding="2" width="100%">
      <xsl:for-each select="child::*" >
        <tr><td align="left"><xsl:apply-templates select="."/></td></tr>
             </xsl:for-each>
          </table>
         </xsl:otherwise>
       </xsl:choose>
     </p>
   </xsl:template>
   <!-- array-recursive: end -->
<!-- array: end-->

<!--note: begin-->
<xsl:template match="note">
  <font class="note">
   (<xsl:apply-templates/>)
  </font>
</xsl:template>
<!-- note: end -->

<!-- picture: begin -->
  <xsl:template match="picture">
  <div class="picture">
  <table border="0"><tr><td align="center">
  <img border="2">
      <xsl:attribute name="src"><xsl:value-of select="@src"/></xsl:attribute>
      <xsl:attribute name="width"><xsl:value-of select="@width"/></xsl:attribute>
      <xsl:attribute name="height"><xsl:value-of select="@height"/></xsl:attribute>
      <xsl:attribute name="alt"><xsl:value-of select="@alt"/></xsl:attribute>
    </img><br />
    <xsl:value-of select="@alt" />
    </td></tr></table>
  <xsl:apply-templates />
  </div>
  </xsl:template>
<!-- picture: end -->


<!-- link: begin -->
  <xsl:template match="link">
    <a class="extern" target="_blank">
      <xsl:attribute name="href"><xsl:value-of select="@src"/>
      </xsl:attribute>
      <xsl:apply-templates />
```

```
    </a>
  </xsl:template>
<!-- link: end -->


<!--verse: begin -->
  <xsl:template match="verse">
    <div class="verse" >
      <b><xsl:value-of select="@verse_name" />:</b><xsl:apply-templates  />
    </div>
  </xsl:template>
<!--verse: end -->

<!--novella: begin -->
  <xsl:template match="novella">
    <div class="novella" margin="20%">
      <center>
        <div class="head">
          <xsl:value-of select="@nov_name" />
</div>
      </center>
      <hr />
      <font size="4em">
<xsl:apply-templates  />
      </font>
    </div>
  </xsl:template>
<!--novella: end -->

<!-- statement: begin -->
  <xsl:template match="statement">
    <div class="statement">
    <br/>
      <font class="announcer">
         <xsl:value-of select="@announcer"/>
      </font>:
      <xsl:apply-templates />
    </div>
  </xsl:template>
<!-- statement: end -->

<!-- grapholog:begin -->
  <xsl:template name="grapholog">
    <center>
    <div class="grapholog">
      <center>

<hr />
<xsl:apply-templates select="picture" />
        <table border="0" width="100%" >
  <tr valign="center" align="center" >

  <xsl:for-each select="array[1]/head">
    <td><xsl:apply-templates select="." /></td>
  </xsl:for-each>
  </tr>
  <hr />
  <tr valign="top" align="left" cellpadding="10">
  <xsl:for-each select="array[2]/text">
    <td><xsl:apply-templates select="." /></td>
  </xsl:for-each>
  </tr>
</table>
</center>
      </div>
    </center>
  </xsl:template>
<!-- grapholog: end -->

<!-- ideonomy: begin -->
  <xsl:template name="ideonomy">
    <center>
      <div class="ideonomy">
<hr />
<xsl:apply-templates select="formal" />
<hr />
<table border="0" width="100%" CELLPADDING="5">
  <tr valign="top" align="left">
  <xsl:for-each select="array[1]text">
    <td><xsl:apply-templates select="." /></td>
  </xsl:for-each>
  </tr>
</table>
      </div>
```

```
    </center>
  </xsl:template>
<!-- ideonomy: end -->

<!-- poetic: begin -->
  <xsl:template name="poetic">
    <center>
      <div class="poetic">
<xsl:apply-templates select="array[1]" />
<hr />
<table border="0" width="100%" CELLPADDING="5">
  <tr valign="top" align="left">
  <xsl:for-each select="array[2]/text">
    <td><xsl:apply-templates select="." /></td>
  </xsl:for-each>
  </tr>
</table>
      </div>
    </center>
  </xsl:template>
<!-- poetic: end -->

<!-- column: begin -->
  <xsl:template name="column">
    <xsl:variable name="length" >
    <!--summen av tekstlengden til 'column', 'header' og 'ingress' -->
      <xsl:value-of select="string-length(concat(head, ingress, text))"/>
    </xsl:variable>

    <center>
    <div class="column">
      <table  border="2" width="75%" valign="top"><tr>
        <td valign="top" width="50%">
  <xsl:apply-templates select="head" />
  <xsl:apply-templates select="ingress" /><hr />
  <xsl:value-of select="substring(text, 1 , ($length div 2) - 500)" />-
  </td>
<td valign="top" width="50%">
          <xsl:value-of select="substring(text, $length div 2 - 499, $length )" />
</td>
</tr>
      </table>
      </div>
   </center>
  </xsl:template>
<!-- column: end -->

<!-- formal: begin -->
<xsl:template match="formal" >
    <div class="formal" >
      <b><xsl:value-of select="@form_name" />:</b>
      <xsl:apply-templates />
    </div>
</xsl:template>
<!--formal: end-->


<xsl:template match="text">
  <div class="text">
    <xsl:apply-templates/>
  </div>
</xsl:template>

<xsl:template match="head">
  <div class="head">
    <p><xsl:apply-templates/></p>
  </div>
</xsl:template>


<xsl:template match="ingress">
  <div class="ingress">
    <p><xsl:apply-templates/></p>
  </div>
</xsl:template>

<xsl:template match="note">
  <font class="note">
   (<xsl:apply-templates/>)
  </font>
</xsl:template>

<!-- et cetera -->
<xsl:template match="i">
```

```
  <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="b">
  <b><xsl:apply-templates/></b>
</xsl:template>

<xsl:template match="br">
  <br />
</xsl:template>

<xsl:template match="hr">
  <hr />
</xsl:template>

</xsl:stylesheet>
```

A.4.2 Cstyle – CSS

```
body{
    color:black;
    background-color: white;
    margin:2em;
    padding:1em
    }
div.column{
    margin:2em;
    padding:1em; border-style:inset;
    border-width:2;
    border-color:gray;
    }

div.set{
    margin:1em; padding:1em;
    border-style:inset;
    border-width:7;
    border-color:gray;
    background-color:eeeeee
    }

div.array{
    margin:3%;
    padding:3%;
    border-style:inset;
    border-width:2;
    border-color:gray;
    div-align:center
    }

.clickhead{
    color: black;
    font-family: fantasy;
    font-style: sans-serif;
    text-decoration:underline;
    font-size:+1.5em;
    text-align:center;
    margin-bottom:1em
    }

.clickhead:hover{
    background:gray;
    color:white;
    font-weight:bold
    }

.hoover:hover{
    background:gray;
    color:white;
    font-weight:bold
    }

.head{
    font-family: fantasy;
    font-style: sans-serif;
    font-size:+1.5em;
    font-weight:bold
    }

div.center_head{
    font-family: fantasy;
```

```
    font-style: sans-serif;
    font-size:+2em;
    font-weight:bold;
    text-align:center
    }

.ingress{
    font-size:+1.25em;
    font-weight:bold
    font-color:black;
    }

.col_text{
    font-size:small
    }

div.graphолog{
    margin:2em;
    border-style:solid;
    border-width:2
    }

div.novella{
    margin:2em;
    border-style:solid;
    border-width:2;
    margin:2em
    }


div.formal{
    margin-top:1em;
    margin-bottom:1em;
    margin-left:4em;
    margin-right:4em;
    font-family: fantasy;
    font-family:serif;
    font-style:oblique;
    font-size:+1.5em;

}

a.extern{
    color:black
    }

a.extern:hover{
    background:gray;
    color:white;
    font-weight:bold;
    }

div.verse{
    margin:2;
    text-align:left;
    font-style:italic;
    font-size:-1
    }

font.note{
    font-family: serif;
    font-style: oblique;
    font-size:80%
    }

.announcer{
    background: silver;
    font-weight: bold;
    font-family:serif;
    color: dark;
    }

picture{
    border-color: gray;
    border-style: double;
    }
```

A.4.3 PHP-Skript for Dynamisk Presentasjon

```
<?php // -*- mode: perl; -*-
```

```
//dekl-argumentet henviser til deklarasjonsfilen
//inst-argumentet henviser til instanseringsfilen

//deklarasjoner
$xsltproc = xslt_create();
$html="tom fil";
$xmlfile = "/tmp/xml_toxmlfile.xml";

//Oppretter fil XML-dokumentet kan skrives til;
$dir = '../Dekleks/';
echo `python2 obdok.py $dir$dekl $dir$inst obdok.dtd > $xmlfile`;

//Kjører xslt-transformasjonen
echo `$less $xmlfile`;
$html = xslt_process($xsltproc, $xmlfile, 'obdok.xslt');

unlink($xmlfile);//Fjerner filen fra tmp-katalogen!
if (!ereg('Mozilla/5', $HTTP_USER_AGENT)) {
//  Må vise hele dokumentet i andre browsere enn Mozilla
    $html = ereg_replace('style="display:none', '', $html);
    $html = ereg_replace('href="cstyle.css"', 'href="alt_cstyle.css"', $html);
    print "<h2>Advarsel: Siden vises best i Mozilla 5.0 eller nyere<h2>";
}
//Bytter spesialtegn til HTMLs character ref. entity
$html = ereg_replace('Aring;', '&#197', $html);
$html = ereg_replace('aring;', '&#229', $html);
$html = ereg_replace("AElig;", "&#198", $html);
$html = ereg_replace("aelig;", "&#230", $html);
$html = ereg_replace("Oslash;", "&#216", $html);
$html = ereg_replace("oslash;", "&#248", $html);
//Bytter tilbake alle forekomster av ÆØÅ

// Detect errors
if (!$html) die('XSLT processing error: '.xslt_error($xsltproc));
xslt_free($xsltproc);// Destroy the XSLT processor

// Output the resulting HTML
$file = fopen("/tmp/king", "w");
fwrite($file, $html);
fclose($file);
include("/tmp/king");
?>
```

A.5 onclick

```
// Swap visibility of a an element.
// The element <em>must</em> be a block element.
// Inline or other types is not supported!
function swap_visibility(id)
{
var element = document.getElementById(id)

if (element.style.getPropertyValue("display") != "none") {
       element.style.setProperty("display", "none", "")
}
else {
       element.style.setProperty("display", "block", "")
}
}
```